

USAISEC

AD-A239 068



2

*US Army Information Systems Engineering Command
Fort Huachuca, AZ 85613-5300*

U.S. ARMY INSTITUTE FOR RESEARCH
IN MANAGEMENT INFORMATION,
COMMUNICATIONS, AND COMPUTER SCIENCES
(AIRMICS)

AN ENVIRONMENT FOR SIMULATION OF DISTRIBUTED SYSTEMS

(ASQB-GC-90-014)

February 1990

DTIC
ELECTE
AUG 1 1991
S B D

DISTRIBUTION STATEMENT A

Approved for public release;
Distribution Unlimited

115 O'Keefe Bldg
Georgia Institute of Technology
Atlanta, GA 30332-0800



36 91-06546



91 06546 039

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-188
Exp. Date: Jan 30, 1986

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b. RESTRICTIVE MARKINGS NONE		
2a. SECURITY CLASSIFICATION AUTHORITY N/A			3. DISTRIBUTION / AVAILABILITY OF REPORT UNLIMITED		
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE N/A					
4. PERFORMING ORGANIZATION REPORT NUMBER(S) N/A			5. MONITORING ORGANIZATION REPORT NUMBER(S) ASOB-GC-90-014		
6a. NAME OF PERFORMING ORGANIZATION Innovative Research, Inc.		6b. OFFICE SYMBOL (if applicable)	7a. NAME OF MONITORING ORGANIZATION AIRMICS		
6c. ADDRESS (City, State, and ZIP Code) 180 Cook Street, Suite 315 Denver, Colorado 80206			7b. ADDRESS (City, State, and Zip Code) 115 O'Keefe Bldg., Georgia Institute of Technology Atlanta, GA 30332-0800		
8a. NAME OF FUNDING/SPONSORING ORGANIZATION AIRMICS		8b. OFFICE SYMBOL (if applicable) ASQB-GC	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER DAKF11-89-C-0023		
8c. ADDRESS (City, State, and ZIP Code) 115 O'Keefe Bldg., Georgia Institute of Technology Atlanta, GA 30332-0800			10. SOURCE OF FUNDING NUMBERS		
			PROGRAM ELEMENT NO. 62783A	PROJECT NO. DY10-01-01	TASK NO. 07
11. TITLE (Include Security Classification) Environment for Simulation of Distributed Systems (UNCLASSIFIED)					
12. PERSONAL AUTHOR(S) Mohsen Pazirandeh					
13a. TYPE OF REPORT Final		13b. TIME COVERED FROM 6/1/89 TO 12/31/89		14. DATE OF REPORT (Year, Month, Day) 1990 February 15	15. PAGE COUNT 49
16. SUPPLEMENTARY NOTATION N/A					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB-GROUP	Simulation, Distributed Systems, Operating Systems, Data Base Management Systems, Performance Measurement, Optimization		
19. ABSTRACT (Continue on reverse if necessary and identify by block number)					
<p>The purpose of this research was: (1) to study the feasibility of developing an environment for the simulation of distributed systems, and (2) to build a prototype version of the proposed environment to show some of its capabilities. The research showed that such an environment can be developed and must contain a number of libraries including hardware components, operating systems, Data Base Management System, algorithms, performance measures, and several knowledge bases.</p> <p>The capabilities of such an environment was demonstrated via the implementation and assessment of the performance of a distributed database implemented under a DBMS, native operating system, and the Cronus distributed operating system. A somewhat complex set of algorithms for evaluating the performance of the database was developed and implemented. This prototype will form the baseline for Phase II development. The major features and components of the prototype tool which will be expanded during Phase II development are as follows:</p> <p>- The detailed definition and updating of a number of libraries, including operating systems, performance measures, DBMSs, and databases. (continued next page)</p>					
20. DISTRIBUTION / AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED / UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED		
22a. NAME OF RESPONSIBLE INDIVIDUAL Son T. Nguyen			22b. TELEPHONE (Include Area Code) (404) 894-3136		22c. OFFICE SYMBOL ASOB-GC

(continued from Block 19)

- Easy assignment and reassignment of system elements (operating systems, DBMSs, etc.). This is accomplished by selecting the desired elements and "clicking" on the new host.
- The development of a detailed model of the Cronus distributed operating system which will be used in the full scale version of environment.
- A library of algorithms for developing simulation, analyzing the performance, and optimizing the system operation.
- Knowledge bases for the isolation of performance failures to a device and optimal assignment of application functions to processors.
- Workload and application functions can be defined, assigned to various processors, and analyzed. They can be reassigned to different processors of workstations to evaluate the impact on performance.




Accession For	
NTIS GRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification_____	
By_____	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

The purpose of this research is to determine the feasibility of developing an environment for simulating the operations of distributed systems and mapping them into a parallel architecture. The distributed systems will consist of a hardware architecture and an application which in this case is a discrete event simulation. The application is executed under two operating systems: a distributed operating system (Cronus) and a native operating system which is a function of the host processor. From an analysis perspective, the two operating systems are considered part of the application and will be modeled accordingly. The hardware architecture consists of a set of (possibly heterogeneous) processors connected via an interconnection network. The application (simulation and the operating systems) is decomposed into functions and distributed to various hosts. The interprocessor communication among processors is handled by the distributed operating system Cronus while the local requirements are served by the native operating system. There will be heavy interaction between the distributed operating system and the native operating system of each host processor.

This research report is not to be construed as an official Army position, unless so designated by other authorized documents. Material included herein is approved for public release, distribution unlimited. Not protected by copyright laws.

THIS REPORT HAS BEEN REVIEWED AND IS APPROVED

s/ 
John W. Gowens
Division Chief
CNSD


s/ 
John R. Mitchell
Director
AIRMICS

Table of Contents

List of Figures.....	1
1.0. Introduction.....	1
2.0. Phase I Objectives.....	3
2.1. Technical Objective 1. Specification of Environment.....	3
2.2. Technical Objective 2. Proof of Concept.....	4
3.0. Summary of Phase I Results.....	4
3.1. Features of Prototype Tool and the Proposed Environment.....	4
3.1.1. User Interface.....	5
3.1.2. Analysis.....	5
3.1.3. Output.....	6
3.2. Phase I Limitations and Assumptions.....	6
3.3. Assessment of Phase I Research.....	7
4.0. Specification of the Environment.....	8
4.1. Requirements of the Environment.....	9
4.1.1. Characteristics of Distributed Systems.....	9
4.1.2. Characteristics of Simulation Models.....	10
4.2. Elements of the Environment.....	10
4.2.1. User Interface.....	11
4.2.2. System Definition.....	12
4.2.3. CPU Editor.....	19
4.2.4. Operating System Editor.....	20
4.2.5. Data Base Management System Editor.....	21
4.2.6. Database Editor.....	22
4.2.7. Function Editor.....	23
4.2.8. Workload Editor.....	24
4.2.9. Analysis.....	25
4.2.10. Performance Measures.....	26
4.2.11. Library of Algorithms.....	26
4.2.12. Knowledge Base.....	27
5.0. Proof of Concept.....	28
5.1. Assumptions.....	29
5.2. Configuration.....	29
5.3. Analysis.....	30
5.4. Results.....	31
5.4.1. Utilization Summary.....	31
5.4.2. Response Time Report.....	32
5.5. Default Values.....	32
6.0. Distributed Operating System (Cronus).....	33
6.1. Description of Cronus.....	33
6.1.1. Application Layer.....	35
6.1.2. IPC Layer and Cronus Kernel.....	35
6.1.3. Network Layer.....	36
6.2. Model of Cronus.....	36
7.0. Phase II Plans.....	37
7.1. Development of User Interface.....	37
7.2. System Definition.....	37
7.3. Measures of Effectiveness.....	38
7.4. Library of Algorithms.....	38
7.5. Knowledge Base Development.....	38
7.5.1. Optimal Assignment of Functions to Processors.....	39
7.5.2. Knowledge Base for Isolation of Performance Failures.....	41

8.0. Phase I Prototype Software Description.....	43
8.1. Phase I Prototype Overall Class Hierarchy.....	43
8.2. Object Hierarchy for Graphical Elements of the Phase I Prototype.....	46
8.3. Phase I Prototype Editor Windows.....	46
8.4. Phase I Prototype System Architecture Object Description.....	47
9.0. Bibliography.....	49

List of Figures

Figure 1. Multiple roles of design assessment.....	8
Figure 2. Opening window.....	12
Figure 3. Software options.....	16
Figure 4. Software elements	17
Figure 5. Software elements assignment status menu.....	17
Figure 6. System operations menu	17
Figure 7. Algorithm menu.....	18
Figure 8. CPU editor.....	19
Figure 9. Operating system editor	20
Figure 10. Data base management editor	21
Figure 11. Data base editor.....	22
Figure 12. Function editor.....	23
Figure 13. Workload editor	24
Figure 14. Analysis window.....	25
Figure 15. Schematic of updating a record	28
Figure 16. Distributed architecture of the example	30
Figure 17. Cronus protocol hierarchy.....	35
Figure 18. Interface of Cronus, client, and native operating system	37
Figure 19. Flow chart of branch selection technique.....	41
Figure 20. Object hierarchy for graphical elements of the Phase I prototype.....	46

An Environment for Simulation of Distributed Systems

1.0. Introduction

The purpose of this research is to determine the feasibility of developing an environment for simulating the operations of distributed systems and mapping them into a parallel architecture. The distributed systems will consist of a hardware architecture and an application which in this case is a discrete event simulation. The application is executed under two operating systems: a distributed operating system (Cronus) and a native operating system which is a function of the host processor. From an analysis perspective, the two operating systems are considered part of the application and will be modeled accordingly. The hardware architecture consists of a set of (possibly heterogeneous) processors connected via an interconnection network. The application (simulation and the operating systems) is decomposed into functions and distributed to various hosts. The interprocessor communication among processors is handled by the distributed operating system Cronus while the local requirements are served by the native operating system. There will be heavy interaction between the distributed operating system Cronus and the native operating system of each host processor.

The traditional modeling practices and approaches, successfully applied to Von Neumann architectures cannot be easily applied to the present problem. There are a number of reasons for this including:

- Distributed system applications and architectures have unique characteristics whose evaluation requires new tools and analysis techniques.
- The presence of a distributed operating system (e.g. Cronus) and its interface with the native operating system adds additional complexity to the problem. From the parallel system view point, the distributed operating system is an application which competes for resources with the applications it is servicing.
- Simulations possess characteristics which distinguish them from other distributed system applications. Such issues as entities being simulated, random number generators, the generation of statistical distributions, the derivation of confidence intervals, etc. affect the functional decomposition of the application, and the performance of the parallel system.
- The modeling environment will be a parallel system whose performance depends on how the inherent parallelism of the simulation is exploited, how the simulation is decomposed into functions, and how the functions are assigned to processors of the underlying architecture. The treatment of the distributed operating system (Cronus) presents additional complexity since it can be mapped into the parallel system in a number of ways with varied performance consequences. A general discussion of the issues in the design of an application for a parallel system and methods for assessing the quality of the mapping is discussed in Pazirandeh [8].
- The quality of the mapping and the evaluation of two competing mapping strategies are dependent on metrics employed to measure performance. A number of metrics have been introduced for this purpose, but it is generally accepted that no single metric can be applied to all scenarios. Measures to satisfy special situations must be defined, validated, and tracked.
- Though the simulation of distributed systems is the primary interest, the analysis of Von Neumann architectures must also be accommodated.

We have concluded that the best way to overcome these technical issues is to develop an environment which can be used to define the distributed system, design the simulation, decompose the application with maximum parallelism, map its functions into a parallel architecture, and evaluate the performance of the total system. Functionally, the environment will have the following elements:

- Distributed system description. The distributed system will consist of a set of components including processors, memory, communication networks, etc. These resources and their interaction with the system functions and distributed application will be specified.
- Describe the distributed operating system. The distributed operating system (Cronus) will be an important part of the simulation and will affect its performance. Cronus interacts with the native operating system to manage the distributed resources. Evaluation of its performance and its impact on the simulation will be a major function of the environment.
- Design of the simulation. The design of the simulation requires defining simulation nodes and entities to be simulated, specifying statistical distributions, simulating message, routing them to different nodes, etc. The environment must provide tools to perform these tasks.
- Decomposition of the simulation. This function provides for the partitioning of the simulation into functions with maximum parallelism. This will result in the development of a processing "tree" specifying the functional dependencies and synchronization points. Communication among processes can also be specified. Since the simulation can be partitioned in a number of different ways, the characteristics of the resulting processing tree can vary significantly. The environment will enable the user to experiment with various partitioning of the simulation.
- Definition of the system architecture. The main purpose of the analysis is to experiment with various architectural concepts to arrive at the most efficient configuration. This allows the user to define the architecture in terms of primitives. Different configurations can be easily constructed. The primitives will include processors, memory, disk, interconnection network, etc. The actual configuration of these primitives can be changed to allow for the evaluation of various scenarios. Memory can be implemented as shared or fragmented, interconnection network can take many forms including hypercube, star, ring, systolic arrays, etc.
- Mapping strategy. This function will assist in arriving at an optimal mapping strategy to assign the application components (the simulation, the Cronus operating system, the native operating system, databases, the Data Base Management System, etc.) into the desired architecture. It will provide tools for the optimization of performance.
- Specification of system parameters. In performing trade-off studies, it is often required that system parameters be varied to evaluate their impact on performance. Therefore, the environment will allow for the specification and easy alteration of major system variables.
- Library of performance metrics. We have already indicated that the traditional performance metrics are not sufficient for the evaluation of modern systems. Therefore, the environment will provide a library of metrics that can measure the operation and the performance of these systems. A description of each metric with the interpretation of its results will be also included.
- Derivation of performance requirements. Most applications lack an explicit set of performance requirements. Yet, the benefits of using an architecture should be quantified in forms familiar to and consistent with the user's expectations. This will require transforming the user's expectations to quantities such as response time, speedup or other measures.

- Library of algorithms. A library of algorithms for describing the system operation and the application's behavior is required. The library must support the analysis of both Von Neumann architectures and distributed systems. They fall into three categories:
 1. Those used in the traditional design assessment techniques based on queueing networks. These include closed form solutions to M/M/m queues, and approximate solutions and estimates for more general statistical distributions.
 2. The operation of modern computers including distributed systems can not be adequately described in terms of queueing networks alone, and new algorithms must be developed to complement these techniques. These include optimization tools, and performance optimization algorithms, such as techniques for synchronization, scheduling strategies, resolution of deadlock, and parallel algorithms.
 3. Simulation models will require the use of a library of statistical routines. These include the generation of statistical distributions, techniques for gathering sufficient statistics, and algorithms for the analysis of output data, e.g., the derivation of confidence intervals.
- Knowledge Bases. Most system analysis tools produce statistical data which can only be interpreted by an expert. In a large system, even the experts have difficulty isolating the causes of a performance failure. A knowledge base which can provide textual feedback on causes of performance failure is desired. In an earlier research we have shown that it is possible to develop such a knowledge base, and we have developed a prototype tool which contains a number of rules experts use to isolate the causes of performance failure, Pazirandeh [7].

A second knowledge base to assist the user in optimizing the assignment of functions to processors will be also available.

- Output Design. The environment should present the results of analyses and simulations in formats which are understandable to the user and are easy to follow. Thus, the system output should be designed and produced to satisfy the user requirements.
- User Interface. The user interaction with the system should provide for two-way communication and feedback. The system should prompt the user for entering the required data and information.

Our ideas and approach for providing some of these functionalities during the Phase I of this research are discussed in the next few sections.

2.0. Phase I Objectives

Recognizing the need for these functionalities, we proposed to evaluate the feasibility of developing an environment for designing and evaluating the simulation of distributed systems during Phase I of our research. Two technical objectives were identified which best reflected our innovation and whose satisfaction would prove the feasibility of Phase I research. These two objectives were described in our Phase I proposal and are as follows:

2.1. Technical Objective 1. Specification of Environment

The purpose of this objective is to define the specification and document the requirements for such an environment. The research to be performed in Phase I in achieving this objective was to use our earlier experience as the baseline and define the enhancements and additions required to enable the prototype tool to be suitable for developing simulation models of distributed systems, and to map them into a parallel system. These enhancements were to include the following:

- Definition of a set of primitives sufficient to represent multiprocessor systems, especially as they pertain to loosely coupled and tightly coupled distributed systems.
- Definition of a set of metrics to measure their performance and the applicability of each measure.
- Identification of performance optimization algorithms, including techniques for synchronization, scheduling strategies, resolution of deadlock, and parallel algorithms.
- Identification of statistical routines, such as random number generators, generation of various statistical distributions, and the derivation of confidence intervals, etc.
- Definition of the requirements of a rule base to interpret the statistical output and recommend actions to improve performance.

2.2. Technical Objective 2. Proof of Concept

We will show the feasibility of these ideas by demonstrating them in a case history supplied by AIRMICS. We have already demonstrated partial feasibility of some of these ideas by two examples on the prototype tool. One example based on a Von Neumann architecture represents an actual operational system. The second example discusses the design of an optimal pipeline for a vector computer.

3.0. Summary of Phase I Results

The purpose of Phase I of this research is: (1) to prove our innovation by showing the feasibility of the technical objectives set out in the proposal, and (2) to build a simple prototype version of the proposed ESDS to show some of its capabilities. The basic idea is to make the tool menu and icon driven. The menus are provided only at the highest level (i.e. presenting major options), and the lower levels of the tool operation are driven by icons (i.e. user will exercise the lower level options graphically by clicking on the appropriate icons).

The prototype tool possesses features which will make its use easy and extremely friendly. We have fully utilized the power of object oriented languages to make it fully graphical and options driven from the screen via either menus or icons. It is designed such that its full scale development can be accomplished in a number hardware platforms, including IBM compatible or Apple Macintosh. The environment can be developed in a number of software environments including C++, or Smalltalk 80 or V. The prototype tool was developed in Smalltalk 80 on an Apple Macintosh computer with 4 MBytes of memory.

3.1. Features of Prototype Tool and the Proposed Environment

In support of Phase I research a prototype tool was developed. The purpose of this development was to show the feasibility of developing the aforementioned environment and to demonstrate some advanced user interface concepts. We demonstrated the proof of concept via the implementation and assessment of the performance of a distributed database implemented under a DBMS, native operating system, and the distributed operating system Cronus. In addition to satisfying the objectives of Phase I research, the major achievements of the prototype tool was the development of an advanced user interface. This prototype will form the baseline for our Phase II development. We demonstrated the prototype tool and the results of our research to AIRMICS and the U.S. Army Information Systems Engineering Command at Fort Huachuca, Arizona, on January 10, 1990.

3.1.1. User Interface

The user interface consists of a main opening window, through which the user can perform almost all operations including hardware and software specification. Also, different operating systems, Data Base Management Systems, databases, and workload elements can be defined and assigned to different system components. Subwindows are used for defining and editing various system elements and input data. Major subwindows are as follows.

- The CPU Definition window allows the user to define, edit and remove processors. Defining and editing activities causes a temporary CPU Definition window to pop up. This window contains labeled fields representing CPU attributes which the user can fill in or edit. The window lists the currently defined CPUs by name.
- The Operating System window allows the user to define and remove operating systems which may subsequently be assigned to various CPUs. The operating system window lists the currently defined CPUs by name. The distributed operating system Cronus is a standard operating system which may assigned to all processors.
- The Data Base Management System (DBMS) window allows the user to define and remove DBMSs which can also be assigned to the various CPUs. The DBMS window lists the currently defined DBMSs by name.
- The database window allows the user to define and remove databases which can also be assigned to the various CPUs. The database window lists the currently defined databases by name.
- The Function definition window allows the user to define and remove software functions and algorithms which can also be assigned to the various CPUs or workload elements. The Function window lists the currently defined functions by name.
- The Workload definition window allows the user to define and remove workload elements or transactions which can also be assigned to the various CPUs. The Workload window lists the currently defined functions by name.

All windows in the Phase I prototype have a fixed size and can be scrolled up and down. All window contents are dynamically updated after any operation which alters their contents. All interface functions are mouse-driven. Clicking the mouse over any displayed item (text or icon) selects that item as input for operations specific to that item. All such operations are presented to the user in menus and all system functions are initiated through menu selections. The Phase I prototype does not support keyboard redundancy for initiating system functions.

3.1.2. Analysis

The proof of concept is supported by the analysis of a simple distributed system. The application consist of a distributed database with a number of requests for read and write initiated from different processors. The architecture and the workload though quite simple, are non-trivial and include many characteristics of a large distributed system. A realistic model of the distributed operating system Cronus is implemented. The native operating system is assumed to be Unix, though other operating systems (e.g. MVS) can also be accommodated. Several DBMSs were also implemented including Ingress and Adabase. A number of databases representing various functions of an Army installation (e.g. personnel, financial, etc.) are designed and distributed to various processors.

The workload consists of a number of requests originated from different workstations or processors and directed to various databases. The requests can be for different forms of service including read and write operations.

An analytical model of the system based on queueing models was developed and implemented. The algorithms account for contention and queue time at various nodes of the system. In a distributed environment, a request issued from one node of the system accessing a database on another node, affects the performance of both nodes. A realistic and detailed model of this scenario was developed and implemented. All algorithms are presented in section 5.3. of this report.

In analyzing the system, the user can exercise a number of options and evaluate their impact on the performance. These include:

- Ability to reassign a database to different processors.
- Ability to reassign a workload element to different processors.
- Ability to reassign a new native operating system from the library.
- Ability to reassign a new DBMS from the library.
- Ability to alter most of system parameters.

3.1.3. Output

The model computes performance parameters (utilization and average response time) by device, workload element, and total. The results of analysis are reported on screen.

3.2. Phase I Limitations and Assumptions

Phase I research, being only a short term feasibility study, was conducted under many constraints and assumptions. During Phase II these constraints will be relaxed and assumptions eliminated. Some of these include:

- Phase I allows analyses based on analytical techniques. Tools and algorithms required to develop discrete event simulations are numerous, and therefore, have to be considered during Phase II.
- The ability to designate subsystems as macros and use them in the analysis of larger systems is an important feature, especially in analyzing systems with hundreds or thousands of nodes. The development of this capability is postponed to Phase II.
- Interconnection networks have significant impact on distributed system performance. During Phase I of this research delays due to the network were not included in the analyses performed in support of proof of concept. The development of a library of interconnection networks with associated message passing protocols will be a Phase II objective.
- The user interface and ease by which the user can interact with the system are essential to the final acceptability of the tool. Phase I of this research produced only sufficient detail to show our concept of this important component. The capability for defining attributes of various components and details were limited. This limited capability was sufficient for Phase I research, but inadequate for the final product. This capability will be greatly expanded during Phase II.
- A desirable feature, considered for Phase II development, is the ability of renaming system entities and parameters. This capability is unavailable in Phase I prototype.

- The output of the present prototype is a simple screen showing utilization and mean response times. In final form, ESDS will have a sophisticated output design and will provide a number of different reports both on-screen and in hard copy. The output requirements of ESDS and our design will be discussed in Phase II technical proposal.
- The development of two knowledge bases mentioned in the introduction is beyond the scope of Phase I research. We have shown partial feasibility of their development as part of our other research projects. We will briefly discuss our plans for this component of ESDS in section 7.0. of this report, and elaborate further in the technical volume of Phase II proposal.

3.3. Assessment of Phase I Research

Phase I of this research was primarily a feasibility study with technical objectives outlined in section 2.0. Our assessment of this phase of the research is that in addition to satisfying the two technical objectives, we were able to develop additional capabilities and advance user interface concepts. These include:

- During the short Phase I period we were able to develop and implement a somewhat complex set of algorithms for evaluating the performance of a distributed system containing a distributed database subject to a number of read and write requests.
- The prototype tool allows for the detailed definition and updating of a number of libraries, including operating systems, DBMSs, and databases.
- The Phase I prototype allows for an easy assignment and reassignment of system elements (operating systems, DBMSs, etc.).
- We developed a detailed model of the distributed operating system Cronus which will be used in the full scale version of ESDS developed during Phase II.
- Workload can be defined, assigned to various processors, and analyzed. They can be reassigned to different processors or workstations to evaluate the impact on performance.
- Innovative and advanced user interface concepts were introduced. Some of these will be discussed in other sections of this report and will form the baseline for Phase II development of the user interface.
- We demonstrated the superiority of object oriented languages as the environment for developing such tools. Smalltalk 80 was adequate for the Phase I feasibility study. We are uncertain whether this language will be able to support full scale development of ESDS. The determination of an appropriate environment will be a Phase II task.

Our general assessment is that we accomplished more than our basic technical objectives set out for Phase I, and the full development of ESDS is to the best interest of AIRMICS and Innovative Research. In addition to providing AIRMICS with an advanced evaluation tool for distributed systems, ESDS will be a highly desirable tool for use by other segments of American industry. This is fully compatible with the spirit and the intent of SBIR program.

4.0. *Specification of the Environment*

In traditional system analysis, a computer system, and hence its model, consists of three components: (1) the application, (2) the system architecture, and (3) the performance requirements (based on a pre-defined set of performance measures, e.g. utilization, response time, throughput, etc.). Usually, one is provided information about two of the three components and the aim of both analysis and modeling is to draw inferences about the third component. The purpose of the analysis depends on which two components are known. Of the three main purposes of modeling, shown in figure 1, the one to ascertain that a candidate architecture can support the processing of an application under a given set of performance requirements is the most widely used.

Confidence in the results of an analysis is dependent on the level of confidence in the input parameters (application, architecture, etc.), and the robustness of the model. The uncertainties of the model are usually no less than the uncertainties of the input parameters, i.e. there is no need to develop complex models if the confidence in input parameters is low. Therefore, the complexity of analysis closely follows the system life cycle, and models are developed with enough detail to meet the analysis needs. During the requirements analysis and early in the design phase, when the level of confidence in system parameters is low, only high level models with very little detail are produced. The effort at this phase of the development is generally directed at finding the "tall poles" of the system's operation and performance. Analytical models best fit this phase. As the system develops and more details become available about the application and the hardware architecture, thus increasing the confidence in input parameters, more detailed models are developed. Late in the development phase, detailed simulation models are appropriate. Thus, the level of detail of the model directly follows the level of knowledge about and confidence in the system input parameters: the more reliable information available about the system the more faithful models can be and are developed.

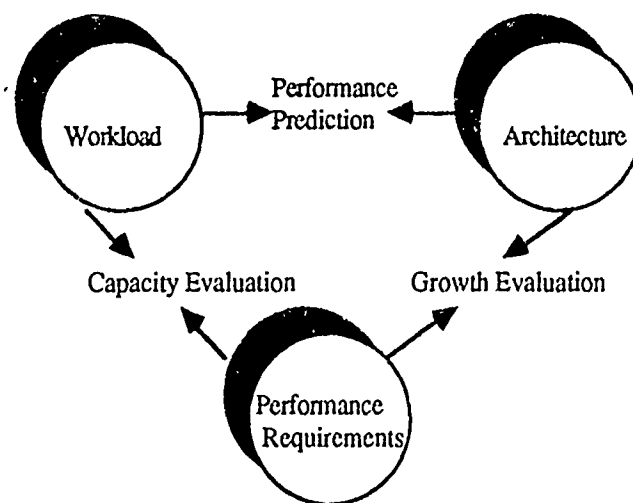


Figure 1. Multiple roles of design assessment

Many applications do not fit any of these three traditional roles of modeling. Some of the reasons include:

- The system architecture for the most part may not always be known. A preliminary analysis may indicate a certain configuration may offer the best solution, yet often a number of uncertainties remain. The major source of uncertainty is that there can be different configurations of the system which may seem adequate to support the processing of the application. Further, one may want to use off-the-shelf components, but their interconnection and communication can vary widely.
- In many cases, the application is known in terms of the functions it will perform with very little knowledge about its timing or sizing. For example in a distributed database application, one may only know the sequence processes a transaction has to go through to produce the desired output. But, the conversion of these processes into software functions with known distribution for insertion into a model is often difficult. Even if the applications is known with some detail, the number of ways it can be decomposed and its functions mapped into the processors are major impediments to applying the traditional modeling techniques.
- The performance requirements of the system are often specified in terms of user expectations rather than in the traditional form of utilization, response time, and throughput. One task of the analyst is to convert these expectations into measurable quantities. In the case of distributed systems, the quantification of user expectations are even more difficult. This is partially due to the absence of general agreement on what constitutes a good set of measures.
- Communication among processors and the synchronization of processes are new parameters, not present in the Von Neumann case. These functions, essentially overhead items, can significantly degrade performance. Further, resource requirements to support these functions are dependent on a number of parameters including the parallelism of the application, mapping of the application functions into processors, processor configurations, and the interconnection network.

4.1. Requirements of the Environment

4.1.1. Characteristics of Distributed Systems

Our ability to develop faithful simulation models of Von Neumann architectures was partially due to the similarity of their organization. This enabled us to represent them by a common set of primitives composed of CPU, memory, LAN, Bus, IO, etc. The advent of new architecture, including multiprocessor systems, has opened up new opportunities and capabilities for solving complex scientific and other resource intensive problems. The principle change from the performance and simulation standpoint is that these architectures and their operations are heterogeneous, and a unified representation is not readily available. These differences can best be illustrated by considering two major classes of multiprocessor systems: loosely and tightly coupled architectures.

Tightly coupled systems are composed of a set of processors which communicate through a shared memory system, and hence there is a high rate of data transfer between each processor and the memory. To improve performance and compensate for the differences in CPU and memory speeds, the shared memory is segmented into several banks. Each processor may also be provided a local memory or cache. The processors are connected to the main memory via an interconnection network, resulting in a totally connected system. As the number of processors increases, the performance degrades due to increased memory contention.

Loosely coupled multiprocessor systems lack shared memory. Instead, each processor is provided an Input/Output device and a large local memory. All interprocess communication and data transfers take place via a message transfer system. The coupling of such systems is quite loose,

and hence the addition of more processor units can be done without affecting the operation of the system. These systems are generally suited for the applications where the communication among processors is minimal, and thus, they are referred to as distributed systems. Because of their inability to support high throughput, loosely coupled systems are not suited for applications which require fast response times, e.g., time-critical real-time systems.

4.1.2. Characteristics of Simulation Models

Simulation models possess unique features and requirements which increase the complexity of the modeling effort. The major technical issues in developing a simulation model, as we discussed in our proposal, are:

- The entities making up a simulation environment can be heterogeneous, and include messages, events, application programs, hardware, or software components. The simulation traffic can be "triggered" for execution in a variety of ways, including message driven, event driven, periodic, and time-queued. Examples of time-queued tasks are software codes which support the simulation of a missile launch. These, while not totally periodic, are time-based. A simulation environment must represent elements that can be present simultaneously.
- From a processor standpoint two different classes of simulation tasks can be present: (1) Tasks whose processor utilization tends to be uniform over time, e.g., steady state simulation, and (2) tasks which have bursty processor demand, e.g., in simulation of command and control environment, a battlefield query may set-off a chain of events with high CPU demand over a short period of time. Depending on the mix of these two, the design and the assignment of tasks, and the scheduling of the processors can vary greatly. Thus, if the application is primarily composed of the first class of tasks, the processors can be scheduled to near full capacity without adversely affecting the performance of the simulation. While, if the application is composed of the mix of the two, the processors must be scheduled for the first class of tasks first, with sufficient excess capacity to handle a second class of tasks and avoid CPU bottlenecks.
- From a memory utilization standpoint, two different simulation tasks can be present. Tasks with a high degree of interprocess communication, generally assigned to a shared memory architecture, and local tasks which have minimal interprocess communication needs. Instruction mix can vary from one application to another. Some codes are highly CPU bound and must be assigned to suitable processors, while others are IO bound and have different needs.
- Simulations usually produce a large amount of data which must be interpreted. In addition, the validity and reliability of statistical experiment is always a cause for concern. We have to ensure that data is gathered in the steady state phase of the simulation, and that the number of replications are sufficient. Thus, a set of mathematical and statistical routines are needed to support these analysis needs. These include random number generators, statistical distributions, methods for gathering sufficient statistics, and the derivation of confidence intervals. Further, techniques for the analysis of a large amount of statistical data resulting from the simulation runs must be developed. These include techniques for filtration, reduction, detection of outliers, and interpretation of data.

4.2. Elements of the Environment

The foremost consideration in developing such an environment is that often the users will not be expert modelers and will rely on the environment to guide them in developing the system model. Being system developers or end-users, they are primarily interested in the performance of the system. The design and the development of the environment should proceed with this in mind, and possess the tools required to support such needs. Flexibility is an important consideration as the environment will be used to analyze a varied class of architectures. Flexibility in this case has a

broad interpretation, covering the ability to define a wide range of architectures, applications, performance measures, and operational concepts (e.g., various types of scheduling strategies, synchronization scenarios, parallel algorithms, etc.). Flexibility is also reflected in the number of options the user will need to design and execute a model. The definition of events, identification of resources to be analyzed, partitioning of the problem, the granularity of the tasks, and their assignments to processors are just a few of these options essential to developing flexible models. The user interface is also an important determinant of its acceptability by the users. We discuss our concept of a friendly user interface in section 4.2.1., and the elements of the environment and a possible design of these elements in sections 4.2.2. through 4.2.12.

4.2.1. User Interface

The user interface will be mouse and menu driven, wherein the user directly manipulates textual and graphical objects whenever appropriate. System data and operation on that data will be selected using a mouse as a pointing device to make selections from lists encapsulated in menus. Menus for choosing data, including CPUs, DBMSs, databases, functions, workload, algorithms, etc. will be screen resident to provide the user with constant feedback about the context of the design and analysis activity. These menus will provide for scrolling when necessary to access items that do not fit into the fixed reserved area. Menus for operators will "pop-up" as a result of user-initiated mouse-button actions on the screen background. Pop-up menus will be context-sensitive, providing for selections appropriate to the context of operation and the area of the screen from which they are invoked. The keyboard will be used as necessary for text input in data definition and to provide shortcuts for menu selections in the context of keyboard-intensive textual input. Also, keyboard redundancy will be provided for many operations to serve those users who prefer this mode of interaction with the scheduling application.

Direct manipulation will be used for scheduling operations on data wherever appropriate. For instance, to link nodes in an architecture hierarchy graph, the user will use the mouse as a pointing device to select node-pairs on the display to indicate the desire to link them. Similarly, data items could be entered into forms, as in CPU definition, by "grabbing" the desired data item from a menu with the mouse and "dragging" an image of the selected item to the appropriate location on the form.

A default windowing environment will be provided for design and analysis as in the Phase I prototype. In addition, it will also be possible for the user to reposition the standard windows to "customize" the screen. These customizations can be saved for future use. Most windows can be moved, resized, and "shrunk" into icons which can then be selected to reopen the window they represent. Editing windows will serve as conceptual "workspaces" in which the menu operations provided will be sensitive to the editing context. We show some concrete examples of these ideas by presenting some specific design ideas of the various windows and options.

4.2.2. System Definition

On start-up, the user is presented with a screen which may be similar to figure 2., with the following options:

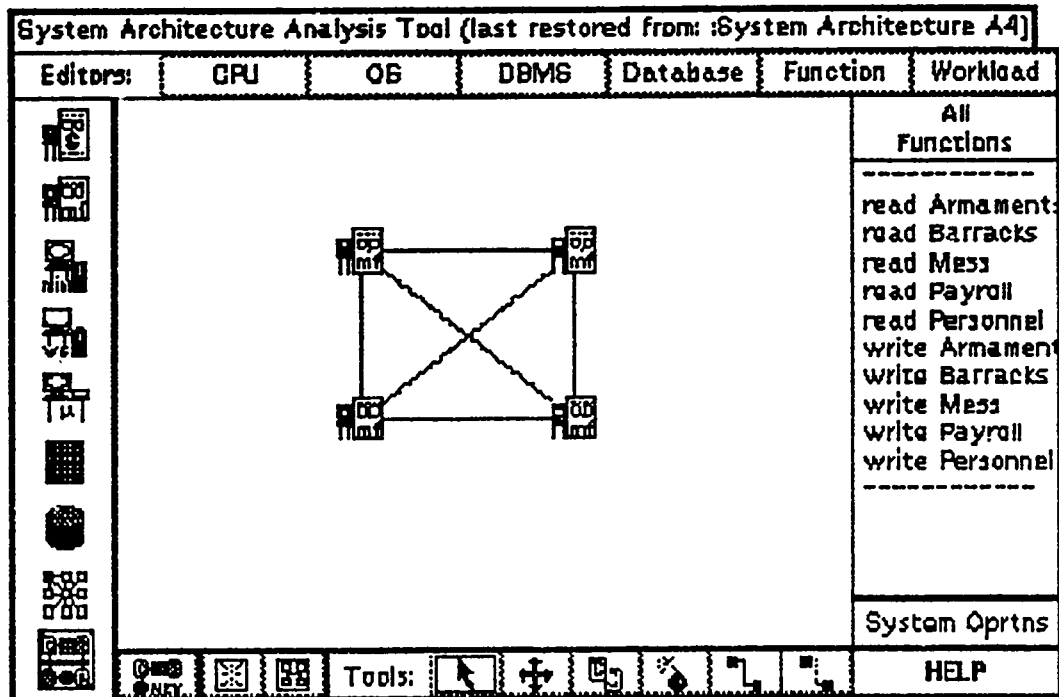


Figure 2. Opening window

The user options accessible from this window fall into the following categories:

1. Editors

The environment contains six editors which are accessed from the corresponding buttons in the main opening window. They are CPU, operating system, DBMS, database, function, workload editors. These editors are discussed in sections 4.2.2. through 4.2.7.

2. Tools

The environment is equipped with the most commonly used hardware elements. For Phase I prototype the following set of icons are provided:

Graph editing tools

The following tools are used to edit the hardware and workload graphs in the corresponding editors. In all cases, the cursor shape becomes the shape of the selected tool. Unless otherwise indicated, the hot-spot of the cursor is the upper left-hand corner.



Arrow Tool.

In the Hardware Architecture Editor, this tool is used to select certain items. In the Phase I prototype, only CPUs may be selected. When selected, the editor for that CPU is activated. In Phase II, this tool will activate an editor for any parameterized hardware element. Additionally, in Phase II, this tool will select a subsystem node which will be expanded to fill the Hardware Graph window.

In the Hardware Architecture Editor, the Arrow Tool is automatically selected whenever the user selects from the list of hardware elements or the list of database, workload, or function elements.

In the Workload Editor, this tool selects the standard background menu.



Move Tool. The move tool is used to move the selected item in the graph.

The user must click and release the mouse button over a highlighted item to select it for movement. The item is moved with the mouse button up. A subsequent click of the mouse button will place the item at the new location. Connections will move dynamically with the item.

In Phase II, we will add a provision for moving groups of nodes in a single operation.

The hot-spot of the associated cursor is the center.



Copy Tool. The copy tool is used to copy the highlighted item.

Copying copies all parameter values to the new instance. Copy functionality is not implemented in the Phase I prototype.



Delete Tool. The delete tool is used to delete the highlighted item.

Confirmation is required. Any connections to the selected item are deleted with the item. The START and END nodes in the Workload Editor cannot be deleted.

In Phase II, we may add a provision for deleting groups of nodes in a single operation.



Connect Nodes Tool. This tool is used to connect nodes to form graph arcs.

The source node is selected by clicking the mouse button over a highlighted item. Then, as the mouse is moved around the window, a "rubber-band" line (anchored at the center of the source node) tracks the mouse. When the mouse button is clicked over a second highlighted item, the destination node, a new link is created and the operation is completed. If the user clicks the mouse while no item is highlighted or if the user moves the cursor outside of the window, the operation is aborted.



Disconnect Nodes Tool. This tool is used to disconnect nodes in a graph.

The source node is selected by clicking the mouse button over a highlighted item. Then, as the mouse is moved around the window, only destination nodes, i.e., those items connected to the source node, will highlight. While a destination node is highlighted, the corresponding connection will be a dashed line instead of a solid line. When the mouse button is clicked over a destination node, the connection to the source node is deleted and the operation is completed. If the user clicks the mouse while no item is highlighted or if the user moves the cursor outside of the window, the operation is aborted.

Graph Editing Buttons:

The following buttons are used to support editing of the hardware and workload graphs in the corresponding editors. Unless otherwise indicated, the action of the button is immediate.



Restore Display Button.

Occasionally, the depiction of the graph may become corrupted. This button forces re-display of the graph.



Clear Display Button.

In the Phase I prototype, this button clears the display of the graph. This is not a useful function and this button will be removed in Phase II.



Subsystem Navigation Button.

In the Phase II Hardware Architecture Editor, this tool will be used to navigate between subsystems. (See the comments on Subsystem Elements in the section on the Hardware Elements List.) When this button is selected a pop-up menu will appear with the following possible selections: TOP, UP ONE LEVEL, OTHER SUBSYSTEM 1, ... , OTHER SUBSYSTEM N. Selection from this menu has results as follows:

TOP displays the top level subsystem. This selection is only available when the user is not already viewing the top level subsystem.

UP ONE LEVEL displays the subsystem that contains the one currently displayed. This selection is only available when a subsystem instance (as opposed to a subsystem template) is displayed and that instance is not TOP.

OTHER SUBSYSTEM X displays the named subsystem.

In all cases, the subsystem currently displayed is replaced by the selected one.

No functionality is implemented for this button in the Phase I prototype. Also, in the Phase I prototype, this button is incorrectly labelled NEW. It should be labelled NAV.

3. Hardware Elements

The following items are in the Phase I prototype Hardware Elements List along the left side of the Hardware-Architecture Editor. The user may select one of these elements by clicking on it in the list and releasing the mouse button. Then, when the cursor is moved into the Hardware Graph Editing window, the shape of the selected element tracks the cursor and will be deposited in the graph at the location where the user next clicks the mouse. Elements must be named when deposited. Subsystem elements must be identified or named when deposited.



CPU Elements.

The Supercomputer, Mainframe, Minicomputer, Workstation, and Microcomputer elements each provide a set of values for default initialization of parameters. Parameter values may be modified via the CPU editor. The CPU editor for a particular CPU is activated by selecting the CPU on the graph using the arrow tool. A CPU editor listing all CPUs by name can be activated using the CPU Editor button at the top of the System Architecture Analysis Tool window.

In Phase II, we will allow the user to define new elements for this list, along with the corresponding default values. Additionally, in Phase II we will provide options that allow the user to modify the parameter values of all instances derived from a particular template by modifying the template. Our plan is to develop a library of computers by the manufacturer and categorized as supercomputer, mainframe, mini-computer, micro-computer, and workstation. Thus, choosing any of these icons will produce a menu of computers available in the system, identified by the manufacturer and model number. For example, an item under "mainframe" may be "IBM 3381". The user can choose a desired computer with a given model name and the system will obtain all its characteristics (speed, operating system, memory, etc.) from the library and insert them in the proper location in the system and use the data appropriately. Thus, the user will not need to enter the detailed characteristics of a computer. The user, of course, is the final authority and can alter the data supplied by the system. Similar options are planned for the storage devices and interconnection network.



Memory Element.

This element designates a memory unit which may be shared by multiple CPUs. In Phase II, memory elements will be parameterized and will be considered in system analysis.



Disk Element.

This element designates a disk drive or disk cluster that may be shared by multiple CPUs. In Phase II, disk elements will be parameterized and will be considered in system analysis.



Network Element.

This element designates a network. No functionality is provided in the Phase I prototype. In Phase II, it will consist of a library of interconnection networks and Local Area Networks (LANs) with associated network protocols. The characteristics of the networks will be implemented parametrically. The user can choose a network from the library, make the desired changes to its characteristics, and use it in the analysis of whole system. The development of the library, like other libraries, will be incremental.



Subsystem Elements.

Subsystem elements have no functionality in the Phase I prototype.

In Phase II, subsystem elements will be defined as named templates. Templates may be defined in one of two ways.

1. By indicating an area of the current Hardware Graph. All elements and connections within the indicated area will be included in the template. Any software assignments to selected hardware will not be included as hardware parameter values in the template.
2. By selecting NEW when an instance is being created (see below).

Instances of the templates will be created whenever the user places a new subsystem element in the Hardware Graph window. At that time, the user will be asked to choose from a pop-up menu where the choices are NEW, and the existing named templates. When a template is instantiated, it must be named as a subsystem.

Subsystem element instances may be modified individually or via global modification to the template.

It is unclear as to how connections to the outside of a subsystem at one level will relate to nodes within the subsystem at the next level. We will investigate solutions to this problem in the Phase II design process.

A subsystem can be designated as a macro and used as a (customized) new icon in a larger system. Macros or subsystems can be nested. Each subsystem can be treated as a separate system, analyzed, and the results passed to the larger system. The set of all macros is kept in a library which can be accessed, individually reviewed, edited, or removed.

4. Software Elements

The software elements fall into three categories, one visible at any given time. A typical option will look like figure 3.

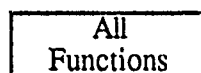


Figure 3. Software options

Clicking on the function will produce a menu of options available to the user (figure 4). Any of the three elements of software can be brought to the foreground by selecting it. Thus, the contents of the visible portion of the software component can be changed as desired. Once a component is visible, its elements can be viewed, edited or assigned to hardware devices. The assignment is accomplished by selecting the desired element, "dragging" it to the host device, and "dropping" it into the device. Selecting any element of the software element (a workload, a function, or a database) will highlight its host device.

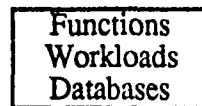


Figure 4. Software elements

Clicking on the "All" option on the menu will present the user the menu shown in figure 5. Choosing "All" will show all elements of the chosen category (workload, function, or database). "Assigned" will show all elements assigned thus far, "Unassigned" will show those not yet assigned, and "Partially assigned" will show those which have been assigned to a subsystem or a macro, but not yet assigned to a specific device. The implementation of these features is planned for Phase II.

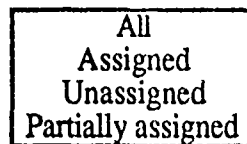


Figure 5. Software elements assignment status menu

5. System Operations

The system operation option allows the user to access all systems, analysis techniques, algorithms and output options. Selecting this will produce a menu of options as shown in figure 6.

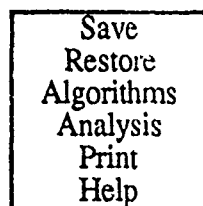


Figure 6. System operations menu

"Save" option saves the working copy of the system to the disk, "Restore" option produces a list of systems saved from previous analyses, any of which can be chosen and opened. "Print" option produces a number of print options including production of hardcopies, and "Help" (when implemented) will provide on-line help. "Algorithms" option provides access to all classes of

algorithms. The algorithms are envisioned to fall into three categories (figure 7). These are further discussed in section 4.2.10. of this report.

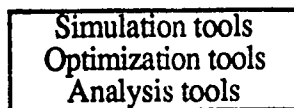


Figure 7. Algorithm menu

4.2.3. CPU Editor

Selecting the "CPU" button from the opening window will produce the CPU editor (figure 8). The pane on the right side of the window contains a list of CPUs defined so far (which can be scrolled if they extend beyond the visible part of the pane). An existing CPU can be reviewed and/or edited by selecting its name from the pane. This action will highlight the CPU icon and will show its characteristics. The operating system and the DBMS can be changed by selecting the applicable button next to each item. This action will produce a list of operating systems and DBMSs available in the system. A new operating system or DBMS can be assigned by selecting from this list or a new one can be defined by selecting the new option from this menu. If a new option is chosen, the form for defining the new item is produced which can be completed and added to the library of operating systems or DBMSs as applicable.



CPUs	
Mainframe 11	<div>CPU:</div> <div>Memory (Mb): <input type="text"/></div> <div>Speed (MIPS): <input type="text"/></div> <div>Operating System: <input type="text"/> </div> <div>DBMS: <input type="text"/> </div>

Figure 8. CPU editor

4.2.4. Operating System Editor

Selecting the "OS" button from the opening window will produce the operating system editor (figure 9). The pane on the right side of the window contains a list of operating systems defined so far (which can be scrolled if they extend beyond the visible part of the pane). The characteristics of an existing operating system can be reviewed and/or edited by selecting its name from the pane. This action will highlight the operating system icon and will show its characteristics. A new operating system can be added to the library by selecting "new" from the menu which presents the user a blank form to add the name and other characteristics of the new operating system.

OSs	
Cronus	<div>Operating System: Cronus</div> <div>Supervisor Overhead (%): <input type="text" value="30"/></div> <div>Scheduler Instructions (K): <input type="text" value="100"/></div> <div>Resource Manager Instrs (K): <input type="text" value="110"/></div> <div>Block Handler Instrs (K): <input type="text" value="100"/></div> <div>IO Handler Instrs (K): <input type="text" value="100"/></div> <div>Queue Handler Instrs (K): <input type="text" value="50"/></div> <div>Native OS Usage Instrs (K): <input type="text" value="100"/></div> <div>DB Write Instructions (K): <input type="text" value="100"/></div> <div>DB Read Instructions (K): <input type="text" value="50"/></div>
DOS	
MVS	
UNIX	
VMS	

Figure 9. Operating system editor

4.2.5. Data Base Management System Editor

Selecting the "DBMS" button from the opening window will produce the Data Base Management System (DBMS) editor (figure 10). The pane on the right side of the window contains a list of DBMSs defined so far (which can be scrolled if they extend beyond the visible part of the pane). An existing DBMS can be reviewed and/or edited by selecting its name from the pane. This action will highlight the DBMS icon and will show its characteristics.

DBMSs	
DB2	<div><div>DBMS: DB2</div><div>Instructions/Read (K): 50</div><div>Instructions/Write (K): 100</div><div>Instructions/Update (K): 150</div></div>
Ingres	
Oracle	

Figure 10. Data base management editor

4.2.6. Database Editor

Selecting the "Database" button from the opening window will produce the database editor (figure 11). The pane on the right side of the window contains a list of databases defined so far (which can be scrolled if they extend beyond the visible part of the pane). An existing database can be reviewed and/or edited by selecting its name from the pane. This action will highlight the database icon and will show its characteristics. The system allows for the definition of very complex and hierarchical databases. This can be done by selecting "new" in the first column of the window (collections). This represents a collection or class of databases. Once defined, the user can define subclasses or databases within the collection. This is accomplished by selecting the collection (which causes the name of the collection appear under the "Tables" segment of the window), and choosing "add new" from the menu in the middle column of the window (titled "Tables"). The user can define an unlimited number of databases belonging to this collection. For example in figure 11, the collection "Armaments" contains the data bases "guns", "nuclear", "planes", and "tanks". Selecting any of the other collections will display their corresponding databases. For any database within the collection, the user can define its fields and characteristics (numeric, alphanumeric, logical, etc.). This is done exactly as it was done for defining databases within a collection. For example in figure 11, the database "Armaments" has two fields; "hand guns" and "rifles".

Databases		
Collections	Armaments Tables	guns Rows
-----	-----	-----
Armaments	guns	hand guns
Barracks	nuclear	rifles
Mess	planes	-----
Payroll	tanks	
Personnel	-----	

Quit		

Figure 11. Data base editor

4.2.7. Function Editor

Selecting the "Function" button from the opening window will produce the function editor (figure 12). The pane on the right side of the window contains a list of functions defined so far (which can be scrolled if they extend beyond the visible part of the pane). An existing function can be reviewed and/or edited by selecting its name from the pane. This action will highlight the function icon and will show its characteristics. The database the function is acting on and the operation it is performing can be changed by selecting the applicable button next to each item. This action will produce a list of databases and operations available in the system. A new database or operation can be assigned by selecting from this list or a new one can be defined by selecting the new option from this menu. If the new option is chosen, the form for defining the new item is produced which can be completed and added to the library of databases or operations as applicable. A new function can be defined by selecting "new" from the menu which results in the presentation of a blank form to be filled by the user. Upon completion and acceptance by the user, the new function is added to the library and its name is displayed (alphabetically) among the other functions.

Functions	


read Armament	
read Barracks	
read Mess	
read Payroll	
read Personnel	
write Armamen	
write Barracks	
write Mess	
write Payroll	
write Personnel	

Function: read Armaments

Block Count:

Instruction Count:

I/O Count:

Database: 


Operation: 

Figure 12. Function editor

4.2.8. Workload Editor

Selecting the "Workload" button from the opening window will produce the workload editor (figure 13). The pane on the right side of the window contains a list of workloads defined so far (which can be scrolled if they extend beyond the visible part of the pane) and all the functions defined on the right hand pane of the window. An existing workload can be reviewed and/or edited by selecting its name from the pane. This action will highlight the workload icon and will show its characteristics on the top and its functional sequence at the bottom of the window. A new workload element can be added by selecting "add new item" from the menu which opens a blank workload definition window. After entering the name and other characteristics of the new element, the user can define its functional sequence. This is done by selecting a function from the function pane, dragging it to the drawing pane, and depositing in the pane. the software sequence can be specified by connecting the function using the "connect" tool. Complex software sequences including branching points with probabilities and synchronization points can be specified. Once the definition of the new workload element is completed and is accepted by the user, its name is displayed in the workload pane.

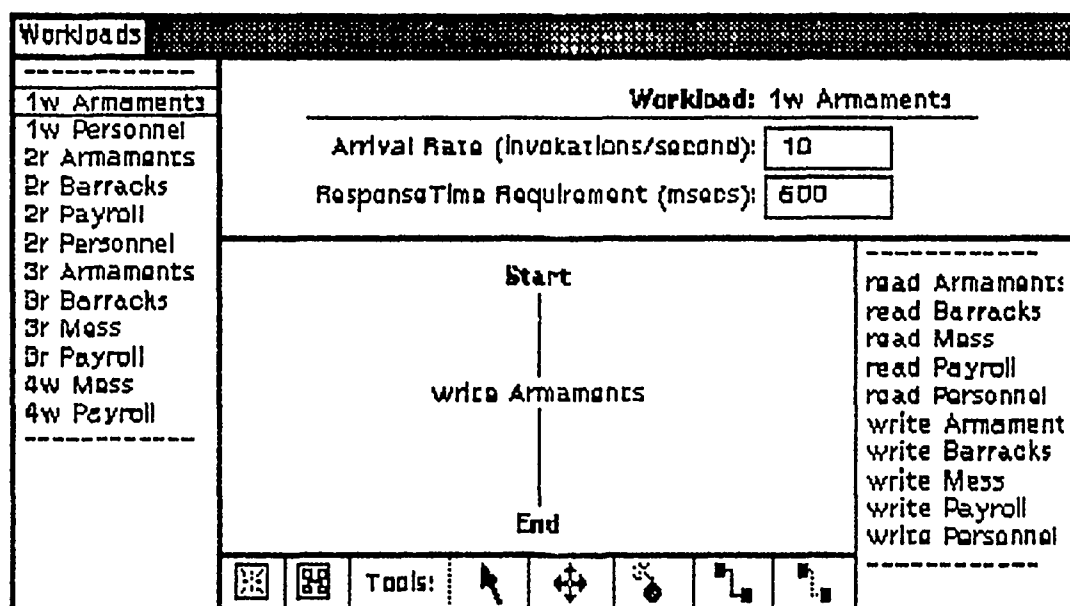


Figure 13. Workload editor

4.2.9. Analysis

Once the system is completely specified, it can be analyzed using the analysis algorithms provided by the system. The Phase I prototype tool contains algorithms for analytical algorithms based on the queueing networks. For this study we have assumed a Markovian process with a First-Come-First-Served queue discipline. The analysis routines are invoked by opening the system operations menu and selecting the "Analysis" option from it. This action will open the analysis window (figure 14). Choosing "perform analysis" from the bottom of this window will open up possible options available to the user, e.g. applying different algorithms or the analysis of subsystems. For this version of the model two options using different algorithms are available. The two versions differ in how they treat a workload element which is initiated from one node in the system and accesses a database located at another node of the system. In one version all CPU usage is allocated to the processor containing the database and in the other the usage of resources are allocated based on the tasks each processor will perform (a more realistic model).

Once the analysis is completed the results are displayed on screen which can be scrolled up or down. Phase II will provide for more sophisticated and detailed reporting of output including the production of hardcopies.

Analysis Results	
Considering database hosts in analysis...	
Utilization summary:	
<hr/>	
rho(Mainframe 11) = 11.08%	
rho(Mainframe 12) = 45.52%	
rho(Mainframe 13) = 30.19%	
rho(Mainframe 14) = 66.98%	
<hr/>	
Workload	Response time
<hr/>	
1w Armaments	7.49766e-4
1w Personnel	0.00374883
2r Armaments	0.0123816
<hr/>	
Clear	Perform Analysis
Quit	

Figure 14. Analysis window

4.2.10. Performance Measures

The traditional metrics of response time, utilization, and throughput, at least in their traditional roles, are neither applicable nor sufficient for assessing the performance of distributed systems. The problem becomes even more complex if one considers different architectures for different implementations of the same application. In addition to the traditional performance metrics, two other classes of metrics are available with broad applicability: application and system dependent measures. Examples of the former are Hockney's $N_{1/2}$ and r_{∞} , and an example of the latter class is speedup. These and the traditional metrics will be the minimum set of measures provided in the environment.

4.2.11. Library of Algorithms

The environment will contain a library of algorithms and routines required to support various analysis needs of the user. The development of the library of algorithms is a Phase II objective. The algorithms can be categorized in three classes:

Library of analysis tools

This is the class of routines which will be used to compute the performance parameters. It include such routines as queueing networks which have been successfully used for the analysis of Von Neumann architectures. At this time, the routines considered for inclusion in this portion of the library are the following:

- Formulas for the derivation of the performance parameters based on Markovian distributions (i.e. M/M/m queues) with various queue discipline (First-Come-First-served, priority queues, etc.). The formulas will include the calculation of utilization, mean and percentile response times, and throughput, by device, workload element, etc. (as applicable).
- Approximate solution for non-Markovian statistical distributions. A number of good approximations are available in the literature which should also be included in the library.

Library of statistical routines

Since the development of simulation models will be a major use of the environment, there will be frequent needs for accessing statistical routines. These include the generation of random numbers, production of statistical distributions with given parameters, and the statistical analysis of the model outputs. The latter requires statistical routines such as the derivation of confidence intervals. Therefore, the library should contain the following statistical routines and algorithms:

- Routines for the generation of uniformly distributed random numbers.
- Routines for the generation of random numbers with given statistical distributions including Poisson, negative exponential, normal, Ehrlangian, geometric, etc.
- Algorithms for distinguishing transient and steady state of a simulation.
- Algorithms for finding the number of replications needed to attain a level confidence of the simulation.
- Algorithms for determining the length of the simulation to gather a sufficient set of statistics.
- Algorithms for the calculation of the confidence intervals.

Optimization tools

Several classes of algorithms will be needed to design an optimal system. These will include:

- Mapping of simulation functions into parallel systems

The major objective in designing a parallel system is to take full advantage of an application's parallelism by decomposing it into parallel functions, and assigning them to processors so that the processors have high utilization with very small variance. At the same time no processor's utilization can exceed one, i.e.; processors cannot be scheduled beyond their capacity. Attempting to achieve uniform utilization among processors may reduce useful work by increasing the communication and other overhead requirements. The major design decision, therefore, is how to assign the functions within the application to processors so that the following two sub-objectives are satisfied:

Subobjective 1. The load among processors is balanced.

Subobjective 2. The communication and other overhead are minimized.

Clearly, these two subobjectives cannot be satisfied simultaneously, and hence, there will be a trade-off between maximizing the load balancing and minimizing the total overhead. Further, any solution to this problem will have to offer the designer the ability to quickly evaluate different assignments and choose the one that best satisfies his specific requirements. It can be shown that the trade-off between the two design objectives can be achieved by:

- Formulating the problem in a (non-linear) optimization context, and
- Developing a technique that finds a "better" solution if it is presented with an existing one which may not necessarily be optimal.

We have shown that this can be formulated as a non-linear optimization problem and solved using a variation of branch-and-bound technique and produce a near-optimal solution, Pazirandeh [8].

- Other algorithms

These algorithms are those required for the description and operation of multiprocessor systems. These include optimization techniques, interprocess communication and synchronization, deadlock resolution techniques, scheduling strategies, and parallel algorithms for multiprocessors.

4.2.12. Knowledge Base

The environment should contain a number of knowledge bases to assist the user in defining the system and interpreting the results of analyses. These fall into three areas:

- A knowledge base to isolate performance failures of the system to a specific component. The development of this knowledge base is a difficult and complex task, and should be a continuing and evolutionary effort. By the conclusion of Phase II, the environment should contain a knowledge base to isolate the cause of a failure to a device or subsystem. Lower level isolation should be part of the further enhancement of the knowledge base. We have shown the feasibility of developing such a knowledge base in an earlier research and will draw upon its results to develop it, Pazirandeh [6,7].
- A knowledge base which will provide feedback to the user on the performance of the system and the quality of mapping the application into the processors. The knowledge base will analyze the assignment and will suggest possible reassignments which will improve performance,

throughput, or speedup. We have also shown the feasibility of developing this knowledge base, and will draw from the results of that research to develop it, Pazirandeh [8].

- A knowledge base to assist the user in designing the system with the correct features, e.g. checking for the integrity of input data is also desired. This knowledge base should be developed in consultation with end-users to identify and document their specific requirements.

Our approach for developing the first two knowledge bases will be further discussed under Phase II plans (section 7.0.).

5.0. Proof of Concept

We demonstrate the capabilities of the prototype tool by analyzing a set of databases consisting of a set of files of Army personnel and related information. We assume that the information about the staff is contained in several records (figure 15). The files containing the different records are assigned to several processors. The updating of data on one record results in changes to all related records. For example, if a staff member is promoted to a new rank, the base pay is automatically updated on the related record. The other records are located on other processors, and the updating is performed via the Cronus operating system.

Two classes of operations can be performed on the databases: operational and informational. The operational consists of those operations which will alter the contents of the database, one of its files, or a record. Operations such as updating a record, writing to a file, adding a new record, and editing a record fall into this category. The informational operation consists of those operations which will not alter the contents of the database, but will require accessing and making queries to its files and records. Operations such as reviewing a personnel file, or making statistical analysis will fall into this category.

When a request is issued, whether operational or informational, the Cronus operating system will process the request, use the native operating system to send the request to other processors. The native operating system on the receiving processor will hand the request to Cronus which will make requested changes or retrieve the desired data from the targeted record (figure 15).

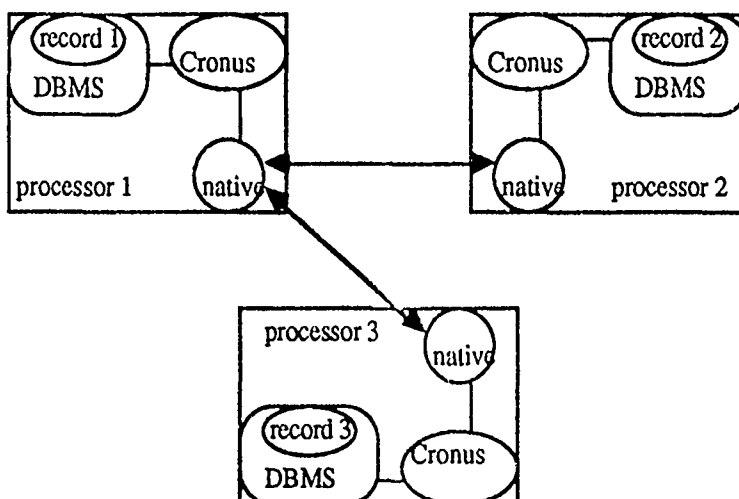


Figure 15. Schematic of updating a record

The purpose of the prototype tool is to show how the eventual environment will perform analysis of the above operations and derive performance metrics (utilization, response time, etc.). We assume that a set of requests have been issued consisting of a mixture of operational and informational requests. We allow the user to specify how many of each operations are to be performed and how the data is distributed across the system. We define the processing steps and assume a (parametric) resource utilization for each request, and use the information to develop a model to predict the system performance under the given scenario.

5.1. Assumptions

For the purpose of this demonstration only, we have assumed the following operational constraints and scenarios:

- The native operating system is UNIX.
- The DBMS is a generic one performing only read and write operations.
- Requests within the same processor will use DBMS, and UNIX
- Requests to other processors will use two calls to DBMS (one in each processor), two calls to UNIX (one in each processor), and one to Cronus.
- UNIX uses $u\%$ of resources.
- Cronus usage is as follows:
 - Supervisor: $csup\%$
 - Interface with UNIX: cu instructions.
 - Interface with DBMS (for write): cdw instructions.
 - Interface with DBMS (for read) : cdr instructions.
- DBMS usage is as follows:
 - DBMS write: dw instructions.
 - DBMS read: dr instructions.

5.2. Configuration

The architecture is a distributed system with the following characteristics (figure 16):

- The system consists on n homogeneous processors denoted by $p(i)$, $i = 1, \dots, n$. Non-homogeneous processors can be easily accommodated.
- Memory response time is not considered for this part of the analysis.
- Data bases are denoted by $DB(i,q)$, where $i = 1, \dots, m$ and q is the host processor.
- Workload elements are denoted by $W(i,F)$, where i is the requesting processor, $F = F(k, DB(j,q))$ is the function it is performing. The function is specified by k , where $k=W$ (write) or R (read), and $DB(j,q)$ is the database it is accessing.

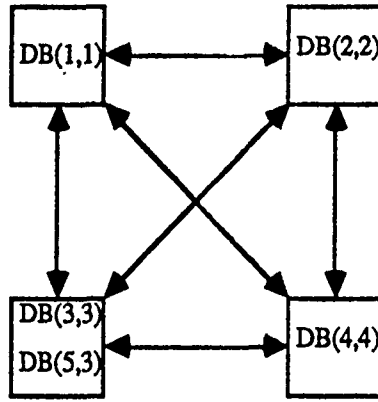


Figure 16. Distributed architecture of the example

5.3. Analysis

- Interarrival calculations

The arrival of traffic to each processor is calculated as follows. For each processor k , $k = 1, \dots, n$, define four classes of requests; write requests within the same processor ($A(k,1)$), read requests within the same processor ($A(k,2)$), write requests to other processors ($A(k,3)$), and read requests to other processors ($A(k,4)$). Thus the four quantities can be calculated as follows:

$$A(k,1) = \sum_{i=q=k} \sum_{F=W} W(i,F,DB(j,q))$$

$$A(k,2) = \sum_{i=q=k} \sum_{F=R} W(i,F,DB(j,q))$$

$$A(k,3) = \sum_{i=k,q \neq k} \sum_{F=W} W(i,F,DB(j,q)) + \sum_{i=k,q \neq k} \sum_{F \neq W} W(i,F,DB(j,q))$$

$$A(k,4) = \sum_{i=k,q \neq k} \sum_{F=R} W(i,F,DB(j,q)) + \sum_{i=k,q \neq k} \sum_{F \neq R} W(i,F,DB(j,q))$$

Then the interarrival time to the k th processor, $\lambda(k,j)$, is given by:

$$\lambda(k,j) = 1/A(k,j), j = 1,2,3,4$$

- Service time calculation

The service time of the k th processor due to the j th type request, $\mu(k,j)$, is computed by :

$$\mu(k,1) = dw / ((1 - u) * MIPS * 10^6)$$

$$\mu(k,2) = dr / ((1 - u) * (MIPS * 10^6))$$

$$\mu(k,3) = (4 * dw / (1 - u) + 2 * cu + 2 * cdw) / ((1 - csup) * (MIPS * 10^6))$$

$$\mu(k,4) = (4 * dr / (1 - u) + 2 * cu + 2 * cdr) / ((1 - csup) * (MIPS * 10^6))$$

where MIPS is the processor speed.

- Utilization calculation

The utilization of the kth processor due to the jth type request, $\rho(k,j)$, and the total utilization, $\rho(k)$, are given by:

$$\rho(k,j) = \mu(k,j) / \lambda(k,j)$$

$$\rho(k) = \sum_{j=1}^4 \rho(k,j)$$

- Response time

Define $R(k,j)$ to be the response time of the jth type request on the kth processor. Then:

$$R(k,j) = \mu(k,j) / (1 - \rho(k))$$

5.4. Results

After execution, ESDS produces several classes of output describing system performance. For Phase I, at the user option, the output can be received at two different levels of detail. One option shows all intermediate calculations, and the other produces the final performance parameters. The output is shown on screen, and is similar to figure 14. The program is quite fast. On a macintosh IICI, the total execution time is less than 30 seconds. The high level summary output is shown tables 1 and 2.

5.4.1. Utilization Summary

The model computes the utilization by workload element, by function, and by processor. The user has the option to view the output at different levels of detail. At the minimum, the summary shown in table 1 is presented.

<u>Processor</u>	<u>Utilization</u>
Mainframe 11	11.08%
Mainframe	45.52%
Mainframe	30.19%
Mainframe	66.98%

Table 1. Utilization summary

5.4.2. Response Time Report

The model computes the response time by workload element and by function. The user has the option to view the output at different levels of detail. A summary showing the response times by workload elements is presented in table 2.

<u>Workload element</u>	<u>Response time (sec)</u>
1w Armaments	7.49766e ⁻⁴
1w Personnel	0.00374883
2r Armaments	0.0123816
2r Barracks	0.00203933
2r Payroll	0.0123816
2r Personnel	0.0123816
3r Armaments	0.00289904
3r Barracks	0.00289904
3r Mess	4.7749e ⁻⁴
3r Payroll	0.00289904
4w Mess	0.0336538
4w Payroll	0.00673077

Table 2. Mean response time summary

5.5. Default Values

The following default assignments and values are used for initial calculations of performance parameters. The user can change most of these parameters to exercise "what if" questions, including reassigning the databases and the workload elements to different processors

- Number of processors: $n = 4$
- Number of databases: $m = 5$ with the following initial assignments:
 - Data base 1 to processor 1, i.e. DB(1,1)
 - Data base 2 to processor 2, i.e. DB(2,2)
 - Data base 3 to processor 3, i.e. DB(3,3)
 - Data base 4 to processor 4, i.e. DB(4,4)
 - Data base 5 to processor 3, i.e. DB(5,3)
- Workload has 12 elements with the following per second arrival rates:

• $W(1,w, DB(1,1)) = 10$	$W(1,w, DB(5,3)) = 10$	$W(2,r, DB(1,1)) = 15$
• $W(2,r, DB(2,2)) = 15$	$W(2,r, DB(4,4)) = 15$	$W(2,r, DB(5,3)) = 10$
• $W(3,r, DB(1,1)) = 20$	$W(3,r, DB(2,2)) = 25$	$W(3,r, DB(3,3)) = 20$
• $W(3,r, DB(4,4)) = 25$	$W(4,w, DB(3,3)) = 30$	$W(4,w, DB(4,4)) = 30$
- Other system parameters are as follows:
 - $u = 0.25$
 - $csup = 0.30$
 - $cu = 100,000$
 - $cdw = 100,000$
 - $cdr = 50,000$
 - $dw = 100,000$
 - $dr = 50,000$
 - MIPS = 10

6.0. Distributed Operating System (Cronus)

6.1. Description of Cronus

Cronus is an object oriented distributed operating system which interacts with the native operating system. The interaction between Cronus and the native operating system is not a master-slave relationship rather its is a complementary one, in the sense that the application is served by both operating systems based on the type of service requested. Each operating system calls upon the other whenever a request issued by the application program falls into its domain of responsibility. Generally, the requests associated with the distributed environment are handled by Cronus while local processing requests are handled by the native operating system. Sometimes Cronus uses the facilities of the native operating system to perform its own internal tasks. Thus, the interaction between the two operating systems is quite complex. The operations of the operating systems and which one is handling the user's requests are transparent to the application program.

The Cronus operating system consists of a set of "objects". An object can be any entity; e.g. a process, a device, a message, or a traffic element. An object is described by two inseparable characteristics: its state and the operations performed on it. The state of the object describes information about its present status. The operations are initiated by the application programs (called the "clients"). The set of operations collectively determine the object's behavior. Often, objects share similar properties or similar operations are performed on a set of objects. A set of objects sharing similar behavior form an "object class" and each element of the set is called an "instance" of the class. An object always belongs to a class, even if it is the only instance of that class. Each object has its own state called its "instance variable". For example, the class of all CPUs forms an object class. Then characteristics of the CPUs are the state of the class, e.g. CPU speed or its operating system. IBM computers is a subclass of this class and IBM 3861 is an instance of object class CPU. The MIPS rating of 8 and MVS are its instance variables of IBM 3861. A subclass of an object class inherits all properties of its superclass, e.g. the subclass "IBM computers" inherits the properties of having a MIPS rating and an operating system.

We can associate one or more procedures, called a "method", to an object, e.g. to the object class "CPU", we can associate a method for computing the impact of the operating system on performance. The actual implementation of method can vary from object to object, and hence the method for computing the impact of the operating system can vary from CPU to CPU. The important point is that once a method is associated or defined for an object or an object class, it has to be specified for each instance of that object. Thus, specifying the method for computing the impact of the operating system for the class of CPUs requires its specification for each subclass or instance of that class. Methods are inherited by the subclasses of an object class, but we can redefine the method for any subclass. Hence, if the method for computing the impact of operating system is defined for the class of CPUs, it is assumed that the computation is inherited by all subclasses and instances of this class. However, we can insert a new definition of the method for a subclass or an instance of the object.

Cronus treats all resources and entities it manages as objects. The object classes it represents include processes, files, resources, and the elements of an operating system. In the Cronus environment, communication between a client requesting a service, and a server providing the service follows a specific protocol which has great implication on its performance, and consequently, on its model. The communication protocol hierarchy closely follows a set of protocol layers. Each layer of the client communicates with the corresponding layer of the server, called its peer. At the highest layer, the client communicates with the "object manager" of Cronus. However except for the lowest layer, there is no direct data transfer between peers. Instead, each layer of the client and the server passes the control and data informations to the next lower or higher (as appropriate) layer of its own hierarchy. Thus, when a client invokes an operation, the message is passed through each layer of its protocol hierarchy to the lowest layer, is transferred to system side, and is passed up through each layer, ending up at the object manager. The core of the Cronus operating system is the Object Manager (OM). OM is a complex program which performs the bulk of the supervisory functions including message handling, access control, multi-tasking, data storage, and other application programs requests pertaining to the distributed environment. It also interface with the native operating system to use its functions and facilities for servicing local requests. Figure 17, taken from [2], shows various Cronus protocol hierarchy layers which are briefly described in the following sections. A more detailed discussion appears in [2].

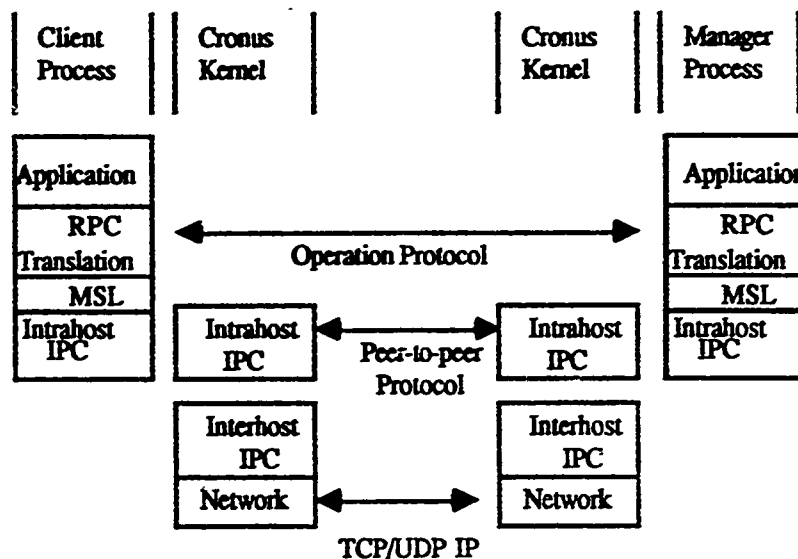


Figure 17. Cronus protocol hierarchy

6.1.1. Application Layer

The application layer consists of entities which fall outside of the domain of Cronus. It is primarily concerned with the programs written by the user and the objects that these programs act upon. Applications invoke actions which will operate on objects (e.g. data sets) and receive feedback from them. The application layer communicates through a subroutine library called Process Support Library which is used for invoking program-generated operations.

• Remote Procedure Call (RPC) Translation Layer

The RPC Translation layer translates the procedure calls within an application layer to an operation invocation on a data object. The RPC layer consists of a library of subroutines within the object manager which link with the application program and are executed as part of the application process. The RPC layer receives operation invocations from processes on other processors via the Inter Process Communication (IPC) layer. It then processes the message and sends it to its destination. Within RPC, operations are performed via Operation Protocol (OP) whose main task is to convert native data types into canonical parameters, transport these machine-independent values among processors, and convert the canonical data back into new host's data types. Acknowledgement messages are sent back taking similar, but the reverse, route. An OP message is an ordered pair (K,P), called key-value pair, where K is the parameter name and P is its value. The key-value pairs are constructed from a set of Canonical data types which are machine-independent data representation. This will allow transfer of data among heterogeneous processors. A library of subroutines called Message Structure Library (MSL) facilitates the transformation of machine- or language-dependent data into Canonical data types.

6.1.2. IPC Layer and Cronus Kernel.

After constructing the message, RPC layer passes it to the next lowest layer in the hierarchy, called the InterProcess Communication layer. This is accomplished through a Cronus routine called *Invoke*. *Invoke* uses the native operating system's IPC to transfer OP messages from the client's domain to a Cronus component called *Cronus Kernel*. The kernel is a Cronus application program which runs under the native operating system on each host. It is composed of five components:

- *Operations Switch* is the standard interface between Cronus and the network software responsible for intrahost message delivery. It locates the manager process capable of handling the request and delivers the message either directly to the process if it is on the same host or to the Operations Switch to host where the manager is located. Operation Switches communicate among themselves using a low level protocol called *peer-to-peer* protocol. This protocol is used for the specification of type of messages transferred and managing the network resources.
- *Locator* is an adjunct to Operation Switch, and using the network broadcast facilities, manages the protocol for locating objects in a distributed environment.
- *Process Manager* resides in the Cronus kernel and shares table data structures with Operation Switch. It converts the Cronus process objects to system calls of the native operating system. It also uniformly defines the unique features of local processes. Thus, it is the major bridge between the Cronus operating system and the native operating system.
- *Request Manager* is used to maintain information about the status of objects across the network, to inform the user of the progress of tasks, and to notify the users if the system has crashed.
- *Broadcast Repeater* is used when a number of interconnected networks are used in a distributed environment. Its function is to maintain communication among the networks which is transparent to the applications.

6.1.3. Network Layer

The Network Layer is the lowest layer of communication protocol. It provides for reliable data transport across the network (not to be confused with Network Layer is different from layer 3 of ISO). The Network Layer supports the Internet protocol including Transmission Control Protocol (TCP), the User Datagram Protocol (UDP), and Internet Protocol (IP).

We will treat the native operating as a generic operating system providing the usual services with a known set of overhead parameters. This method of accounting is a generally accepted method of modeling an operating system, especially when detailed information about the operating system is not available or detailed modeling is not required.

6.2. Model of Cronus

Above discussion and our analysis has resulted in a preliminary model of the Cronus operating system and its interaction with the application and the native operating system. Major elements of this model are briefly discussed here and shown in figure 18. We begin by assuming that a request for service is issued by a client in host A.

- Process Support Library (PSL) routines transform the request into an invoke operation of the Cronus.
- Remote Procedure Call (RPC) translates the operation from a procedure call into a appropriate operation on the object or the data set.
- Cronus's object manager receives the request to determine whether the data is located in the same host or must be retrieved from another host.
- Native operating system is used to communicate with Cronus kernel (via intrahost InterProcess Communication).
- Interhost IPC sends the message over the communication network.
- Interhost IPC on the receiving host receives the message and hands the message Cronus kernel.

- Cronus kernel uses the native operating system to hand off the message to RPC translator.
- Object manager on the receiving host finds the data.
- The reply follows a reverse path.

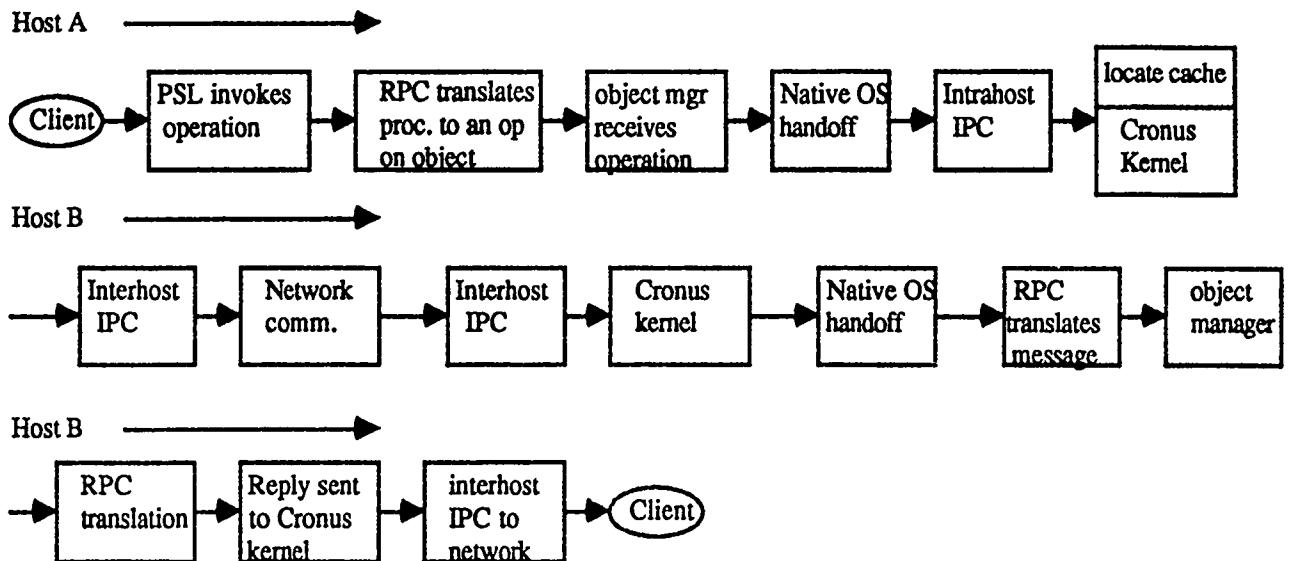


Figure 18. Interface of Cronus, client, and native operating system

7.0. Phase II Plans

We will briefly discuss our plan for Phase II development of the environment.

7.1. Development of User Interface

We have presented our progress thus far and our plans for the development of user interface for Phase II in section 4.2. We will further elaborate on these ideas in the technical volume of Phase II proposal.

7.2. System Definition

The capabilities considered for Phase II development of ESDS will include the following features:

- ESDS will contain a library of all major computers (IBM, DEC, etc.) with their characteristics already defined in the system. The user can choose one by selecting it from a table or define a new one and add it to the library.
- ESDS will contain a library of all major operating systems (Cronus, MVS, VMS, NOS, Unix, Dos, etc.) with their characteristics already defined in the system. The user can choose one by selecting it from a table and assigning it to a processor or define a new one and add it to the library.

- ESDS will contain a library of all major DBMSs (Adabase, Ingress, Oracle, M204, etc.) with their characteristics already defined in the system. The user can choose one by selecting it from a table and assigning it to a processor or define a new one and add it to the library.
- ESDS will contain a library of all major networks (star, ring, etc.) with the associated protocols (IEEE 802, etc.) already defined in the system. The user can choose one by selecting it from a table or define a new one and add it to the library.
- After defining the system the user can choose to apply discrete event simulation or analytical techniques without having to redefine the system all over again.
- ESDS will contain a library of optimization tools and algorithms. These tools can be used to optimize various components of the system, e.g. assignment of functions to processors, developing optimum routing algorithms, etc.
- ESDS will permit for dynamic assignment and reassignment of functions, workload elements, and databases to processor or other elements of the system. This is performed by selecting the item from a menu on the screen, "dragging" it to the desired location, and "dropping" it on its host. This is an important feature of the tool and will be appreciated by those who develop models of computer systems.
- Historically, developing the models of large systems have been difficult and cumbersome. ESDS allows for the definition and the designation of subsystems as macros which can be later used as icons in larger systems, thus avoiding large and unwieldy systems with hundreds of nodes to be analyzed at once. The user can inspect and/or edit a macro by clicking on its icon opening a new window containing its characteristics, and if desired, performing subsystem analysis. The capability of using macros has many advantages which cannot adequately be discussed here.
- ESDS will have an advanced use interface, a sampling of which is discussed in section 4.2. and was demonstrated during our briefing in Fort Huachuca. All user actions will be via a mouse using icons and forms provided by system. Thus the user will never have to be concerned about forgetting to enter the required information.
- Redefining the system elements or characteristics of a component can be done with ease, by pointing the mouse to the desired component or field and entering the change.

7.3. Measures of Effectiveness

It is known that one or two metrics are not sufficient for measuring the effectiveness and the performance of most modern parallel or distributed systems. ESDS will contain a set of performance measures including their description, the area of their applicability, and their interpretation. Some of these measures were discussed in section 4.2.10.

7.4. Library of Algorithms

ESDS will contain the library of algorithms discussed in section 4.2.11.

7.5. Knowledge Base Development

By the conclusion of Phase II development we expect to include the first two knowledge bases discussed in section 4.2.12. One will be a knowledge base to optimally assign the application functions to processors. The second knowledge base will isolate performance failures to a component or a subsystem. We have shown the feasibility of developing these knowledge bases as part of our earlier research efforts. Our plan is to continue the development of these knowledge

bases during the Phase II development. In this section we present a summary of our approach for their development

7.5.1. Optimal Assignment of Functions to Processors

Parallel and distributed systems improve the performance of an application by using its inherent parallelism to process tasks concurrently. An immediate consequence of this is that the application's tasks assigned to different processors may have to pass or share data, exchange messages, or synchronize before another task can initiate processing. These and other services, basically overhead elements, can be resource intensive causing degradation of performance. Therefore, a major concern in the design of such systems is how to proceed with the assignment to attain the maximum speedup over sequential processing. It may appear that this objective can be realized by balancing the load among processors, ensuring that no processor's utilization exceeds one, i.e.; none is scheduled beyond its capacity. However, achieving balanced utilization may in fact degrade performance by increasing the communication and other overhead requirements. The major design decision, therefore, is to devise an assignment strategy that satisfies two objectives: (1) The load among processors is balanced, and (2) The communication and other overhead usage are minimized.

Clearly, these two objectives cannot be satisfied simultaneously, and hence, a trade-off between the two needs to be performed. Further, any solution to this problem will have to offer the designer the ability to quickly evaluate different assignments and choose the one that best satisfies his specific requirements. In this paper we will show that an optimal assignment strategy can be found by: (1) Formulating the problem in a (non-linear) optimization context, and (2) Developing a knowledge base that finds a "better" solution if it is presented with an existing one which may not necessarily be optimal. Our approach briefly is as follows:

Assignment of Value

The application's software flow is decomposed into forks and branches. A branch is a series of tasks to be processed sequentially. A fork is a set of branches emanating from a branching point and converging to a synchronization point. Values are assigned to tasks, branches, and forks reflecting their response times. The value of a task is the sum of the following components:

- Its execution time,
- The queue time due to contention at the processor,
- Memory access or response time,
- Communication time,
- Delay due to synchronization, and
- Operating system overhead.

The value of a branch is the sum of the values of its tasks. The value of a fork is the value of its longest branch. The value of the software flow is the value of its longest fork or branch.

A recursive approach is developed to find a better assignment by improving on a previous one. The improvement is measured as the shortening of the total response time (value) of the software flow. The recursive technique works as follows:

We choose the two branches with the highest and lowest value, and reassign their tasks among processors. If the branch with the highest value is a fork itself, the algorithm is applied to that fork. The algorithm will ensure that the value of the fork will not increase. We begin by considering the branches of the outermost fork in the software flow. In our implementation these algorithms are coupled to a set of rules that will present the results of a run to the user along with the resulting recommendations. The user can interpret the recommendations, make changes as appropriate, and

run the algorithm again. Recommendations guarantee that a faster end-to-end response time (i.e. a smaller value) is produced. The rules are implemented in a knowledge base.

Development of a knowledge base

The problem can be formulated as an optimization problem and solved using a number of AI and operations research techniques. Our approach will be to recursively find a better assignment of tasks to processors by improving on a previous one. The improvement is measured as shortening the total response time of the software flow. We assume that an initial assignment has been made. The technique, in summary, is as follows:

- Choose the branch with the highest value, and balance its load with the branch of the fork with the lowest value. If the branch with the highest value is a fork itself, perform the load balancing with that fork, first. This will result in a fork with lower response time than the one we started with. The user is presented with a recommendation for reassigning tasks which he can reject.
- The load balancing is done by formulating it as an optimization problem.

A top-down approach is used to compute the values of the branches of the outermost fork in the software flow. We apply our algorithm to the longest and shortest branches. The branch selection and load balancing algorithms are presented in detail below. In our proposed implementation, these algorithms will be coupled to a set of rules that will present the results of a run to the user along with any recommendations. The user can interpret the recommendations, make changes as appropriate, and run the algorithm again. The pseudo-code for the algorithm, called 'load-balance', is defined as follows:

Begin load-balance (fork):

1. Compute the value of all branches of a fork.
2. If the branch with the highest value contains a fork, recursively apply the load-balance algorithm to this fork.
3. Choose the two branches with the highest and lowest values for load balancing. Essentially, we are trying to redistribute the load so that longest branch in the load-balanced fork has a faster time than that in the current fork. Hence, the response time of the fork will decrease. Our rationale for choosing the shortest branch to shorten the longest branch is that there is no other branch in the fork with greater 'slack time' available. If the branch with the lowest value (shortest response time) includes one or more nested forks, initially choose the lowest valued branch of these forks. Thus, the response time of the associated fork will probably increase less than if another branch is selected.
4. Load-balance the two selected branches using above algorithm.

End.

After running load-balance on the given architecture, the system will summarize the results and present its recommendations (if any). The user may then redistribute the load as the algorithm suggests and run the algorithm again. The algorithm guarantees that its recommendations will produce a faster end-to-end response time under the given the constraints. We realize, however, that there may be other constraints which are not represented. The user should interpret the recommendations with these other constraints in mind. The algorithm is summarized in figure 19.

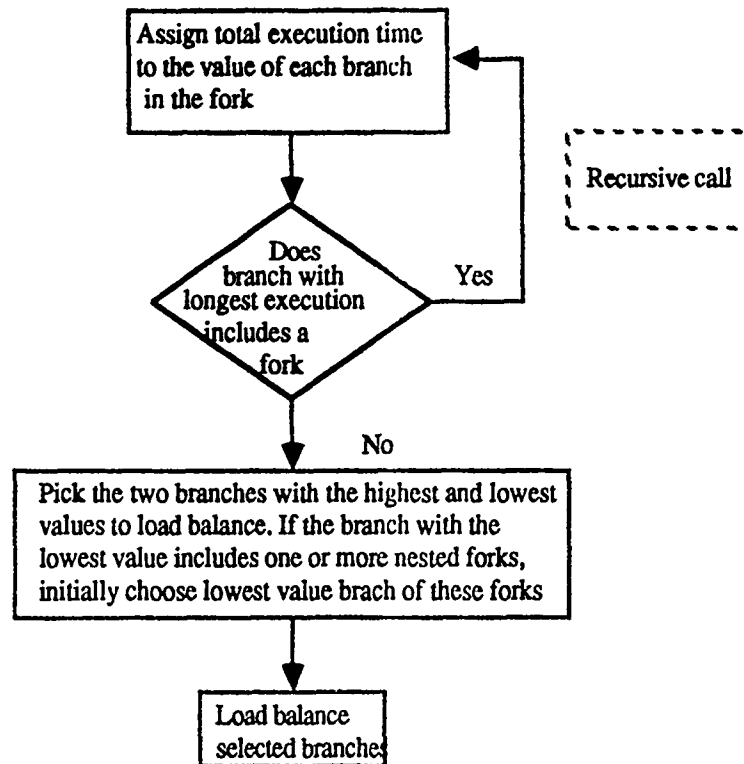


Figure 19. Flow chart of branch selection technique

We have implemented a variant of above algorithm in a combination of an object oriented language and Prolog, and have developed a rule base for designing an optimal pipeline in a vector processor.

7.5.2. Knowledge Base for Isolation of Performance Failures

Of the many roles of performance modeling, the recognition of performance failures and the isolation of their causes are probably the most important. A performance failure is any departure of a system's actual or predicted behavior from its intended (e.g., designed or specified) behavior. Examples of such failure are the inability of a computer system to sustain a desired level of throughput, a job or a transaction failure to meet its prescribed response time, or a device's utilization exceeding a given threshold. A failure can be actual (as in an operational system), or derived or inferred (as predicted by a model of the system). Upon the detection of a failure its causes need to be identified and corrective action taken to ensure the mission success. "Expert" modelers use diagnostic reasoning and data to draw inferences to determine the causes of the failures and take corrective action. The reasons for the absence of such a knowledge base as a part of modeling environments are numerous. Two reasons are thought to be major contributors. First, a performance failure can have numerous sources and its attribution to a specific cause is often difficult. Second, the rules governing the cause(s) of a problem are quite complex and experts often employ subjective and "rule-of-thumb" approach to arrive at the roots of performance failures. Development of a knowledge base for this purpose is a two step process:

- A systematic analysis of a performance problem in the context of total system design and operational concept is required to isolate the failure to a cause.
- The rules leading to the attribution of the problem to a specific cause need to be implemented in a knowledge base.

The seemingly "subjective reasoning" process of finding the causes of a performance failure of a computer system is based on a systematic approach employed by human "experts". It falls into the broad category of problem isolation techniques called Diagnostic Reasoning. The technique has been successfully applied by others to detect and isolate the failures of variety of systems. For example, Davis R.[4] has applied it for the troubleshooting of electronic digital circuits, Breuer, M.A. and Friedman, A.[3] have applied a modified version of the technique for the failure detection using guided probe, and Abramovici, M. and Breuer, M.A. [1] have applied it for cause-effect analysis.

The ideas and the terminology of Diagnostic Reasoning have natural counterparts in computer architecture with possible new interpretations for some. Previous applications of this technique differ from the present problem in an important way. In the past, the technique has been applied to problems and components which have two deterministic states, i.e., components which are either "working" or have "failed". The application we are considering has a stochastic state space, i.e., failure is only a statistical inference, and hence, it possesses a continuum of "working" and "failing" states. Yet, the key Diagnostic Reasoning concepts of structure, behavior, adjacency, causal pathways, and categorization and layering of failures still carry over naturally and can be easily adopted to this problem. Of course, our conclusions and diagnoses will be only statistical, but the same is generally true for problems with deterministic state space, Davis [4].

We see several advantages to this approach for performance diagnosis of computer systems. The most important is impracticality of the alternative approach, namely the development of procedures to test various possible branches. This could entail treating the system as a black box and producing input values with known outputs to test the behavior of various components and subsystems. The data and statistical analyses required to support such an approach are prohibitive. Practical diagnostic techniques based on reasoning and inference do not require such detailed numerical knowledge to be effective. We believe that encapsulated experience, sophisticated "intelligent" search, and qualitative reasoning are the key to practical diagnostic reasoning about systems that have failed their performance requirements.

Secondly, we believe that our knowledge-based approach provides many of the advantages typically claimed for expert systems: transfer of expertise to novices (i.e., providing training as a side effect of the diagnosis process), intuitive support for conclusions and recommendations, and the ability to handle vague and messy problems. Finally, we believe that this line of reasoning leads naturally to insights into system design and design verification.

Principles of Diagnostic Reasoning

One of the standard tools of software engineering and digital hardware is the design of fault detection experiments and diagnosable machines, e.g., based on the construction of homing and distinguishing sequences for finite state machines, Zohari [9]. More recently, a nearly independent set of fault diagnosis theories and programs for digital circuits has been emerging in applied artificial intelligence literature. Stimulated by the seminal contribution of researchers such as de Kleer, Genesereth, and Davis, this line of research has led to demonstration programs - practical for small problems - that rely on intuitively motivated "qualitative reasoning" algorithms and heuristics.

Although the literature on Artificial Intelligence approaches to fault diagnosis and reasoning is rapidly growing and is yet to mature, it has already produced some solid insights and programming techniques that have proved useful in the development of knowledge-based systems for fault diagnosis. Prominent among these are approaches that exploit the concepts of casualty and explanation to drive the process of testing and diagnosis, and to organize computation and search efficiently. Diagnostic Reasoning exploits the knowledge of structure and behavior to draw

inferences about the causes of failures. Structure covers a broad class of entities which make up the system and their interrelationships, including physical components, e.g., hardware devices and their connectivities. Behavior describes the system operation or its specified and expected operation. Behavior is specified by how the system's output relates to its input. A variety of techniques is used to describe behavior, including rules for mapping input to output, Petri-nets, simulation, queuing networks, or a combination of these methods.

The process of Diagnostic Reasoning proceeds by using the structure, behavior, and the symptoms of the failure to generate a candidate set of causes, i.e., determine a set of causes which could have possibly brought about the failure. A fundamental principle in finding the sources of the problem is understanding the way a failure can propagate through the system. This leads to the development of causal pathways, essential to tracing the "path" of a failure. In this effort, one is faced with the inevitable dilemma of complexity vs. completeness, i.e., to produce a complete set of possible causes, a costly and complex search is needed. This dilemma is dealt with by categorizing and layering failures. The essence of layering is the observation that certain failures occur more often than others. Thus, one begins by examining most likely causes and, depending on the desired level of completeness, continue adding less probable causes. This will ensure capturing important causes should one decide to discontinue the search. Hence, associated with each category of failure, there will be a collection of paths of interaction. Thus, an ordering of categories of failures will produce an ordering of paths of interaction and the resulting causal pathways. This will allow constraining candidate generation, while no path is completely eliminated from consideration. We are afforded the opportunity of considering a progressively wider class of candidate causes. The categorization of failures thus obtained will naturally lead to their hierarchical classification.

The discussion demonstrates the importance of pathways in acquiring knowledge and finding the causes of a failure. Implicit in the determination of pathways is the notion of next or adjacent entities, i.e., the causes of failure propagate because the system entities are adjacent. Therefore, the concept of adjacency plays an important and fundamental role in the definition of the pathways of interaction. Adjacency can be physical (standard notion), functional, and/or algorithmic. Clearly, broadening the concept enlarges the number of pathways of interaction.

8.0. Phase I Prototype Software Description

The Phase I prototype is implemented in the Smalltalk object-oriented programming language. Objects are instances of statically defined classes which specify object attributes (instance variables) and behavior (i.e., procedures or methods). Classes are hierarchical and each class inherits all instance variables and methods of its superclass. Methods may be overridden in a subclass by rewriting them.

8.1. Phase I Prototype Overall Class Hierarchy

Below is the complete class hierarchy for the Phase I prototype. In the listing, each tab level represents an inheritance level. For instance, an ActionMenuController is a Controller which, in turn, is an Object. Underlined items are names of classes developed for the prototype. Items not underlined are classes provided by the Smalltalk environment. Quoted strings within parentheses are instance variables. Instance variables are always inherited by subclass objects.

```

Object ()
  Controller ('model' 'view' 'sensor' )
    ActionMenuController ()
    AnalysisButtonsController ()
    MouseMenuController (menus & messages)
      IRGraphController ('form' 'formIs' ... 'savedCursor' 'underIconDisplayMethod' )
      FunctionGraphController ()
      ScrollController ('scrollBar' 'marker' )
      ListController ()
        SelectionInListController ()
          FunctionSelectionController ('redIsYellow')
          RedSelectionInListController ('redIsYellow' )
        ParagraphEditor ('paragraph' 'startBlock' 'stopBlock' ... )
        TextEditor ()
          StringHolderController ('isLockingOn' )
          TextCollectorController ()
          AnalysisController ()
      StandardSystemController ('status' 'labelForm' 'viewForm' )
      ArchBrowserController ()
      SubordinateSystemController ('initialAction' )
      WorkloadBrowserController ()
    DBCcollection ('name' 'tables' )
    DBMS ('name' 'readInstrCount' 'writeInstrCount' 'updateInstrCount' )
    DBTable ('name' 'primaryKey' 'rows' )
    DisplayObject ()
      DisplayText ('text' 'textStyle' 'offset' 'form' )
      Paragraph ('clippingRectangle' 'compositionRectangle' 'destinationForm'
        'rule' 'mask' ... 'outputMedium' )
      IconList ('list' )
    FieldDescription ('promptString' 'longestValueString' 'acceptBlock'
      'initialValueMsg' 'newValueMsg' 'valueClass' )
      IntegerFieldDescription ()
      KIntegerFieldDescription ()
      StringFieldDescription ()
      ChoiceListFieldDescription ('listMsg' )
    GraphArc (really "IRGraphArc") ('element1' 'element2' )
    GraphNode (really "IRGraphNode") ('icon' 'iconSymbol' 'location' 'storageIndex' )
    FunctionalElement ('functionName' )
      ApplFunction ('name' 'blockCount' 'ioCount' 'instrCount' 'dbCallCount' )
      DBFunction ('dbCollection' 'operation' )
    HardwareElement ('name' )
      CPU ('memory' 'speed' 'os' 'dbms' 'applFunctions' 'databases' 'workloads' )
        MainframeComputer ()
        MicroComputer ()
        MiniComputer ()
        SuperComputer ()
        Workstation ()
      Disk ()
      HardwareArch ('elements' 'connections' 'assignedApplFunctions'
        'assignedWorkloads' 'parentArch' )
      Memory ()
      Network ()
    Model ('dependents' )
      ArchBrowser ('systemArch' 'currentSubArch' 'hardwareSelection'
        'softwareType' 'softwareSelection' 'softwareFilter' )

```

'subordinateEditors' 'hardwareTool'
 'activeHardwareElement' ...)
DatabaseBrowser ('dbDictionary' 'collection' 'table' 'row' 'quitting')
FormBrowser ('dictionary' 'formClass' 'newEntryClass' 'entry' 'key'
 'fieldDescriptions' 'entrySelectionView' 'formIOView' 'quitting')
WorkloadBrowser ('functionSelection' 'editingTool' 'activeElement' ...)
StringHolder ('contents' 'isLocked')
TextCollector ('entryStream')
AnalysisCollector ('quitting' 'muStore' 'rhoSumStore'
 'displayIntermediateResults' 'considerOthersInAnalysis')
OS ('name' 'supervisorOverhead' 'schedulerInstrCount' 'resourceMgrInstrCount'
 'blockHandlerInstrCount' 'ioHandlerInstrCount' 'queueHandlerInstrCount')
CronosOS ('nativeUsageInstrCount' 'dbWriteInstrCount' 'dbReadInstrCount')
SystemArch ('hardwareArch' 'applFunctions' 'workloads' 'oss' 'cpus'
 'hardwareElementCounts')
DistributedDBArch ('dbmss' 'dbCollections')
View ('model' 'controller' 'superView' 'subViews' ... 'boundingBox')
ActionMenuView ('labelMsg' 'aspect' 'menuMsg')
ArchToolsButtonView ()
WorkloadToolsButtonView ()
EditorButtonView ()
AnalysisButtonsView ()
DatabaseQuitButtonView ()
DisplayButtonsView ()
NewMacroView ()
FormIOView ('activeFieldView' 'fieldViewWantingControl' 'promptText'
 'formExtent' 'viewsCentered' 'centeringOffset' 'menuMsg')
HardwareGraphView ('activeElement' 'softwareElement' 'lastActiveElement'
 'preMoveLocation')
FunctionGraphView ('functionElement')
ListView ('list' 'selection' 'topDelimiter' 'bottomDelimiter' 'lineSpacing' ...)
SelectionInListView ('itemList' ... 'partMsg' 'initialSelectionMsg'
 'changeMsg' 'listMsg' 'menuMsg')
FunctionSelectionView ()
IconSelectionView ('iconList' 'selectorList' 'iconMsg' 'selectorMsg')
RedSelectionInListView ()
StandardSystemView (... 'minimumSize' 'maximumSize' ...)
ArchBrowserView ()
SubordinateSystemView ('name' 'superOrdinate')
DatabaseBrowserView ()
FormBrowserView ()
WorkloadBrowserView ()
StringHolderView ('displayContents')
FieldView ('active')
ChoiceListFieldView ()
TextCollectorView ()
AnalysisView ()
SubordinateEntry ('name' 'dictionary' 'status' 'view' 'creationBlock')
Workload ('name' 'functionalElements' 'connections' 'arrivalRate' 'responseTimeReq')

8.2. Object Hierarchy for Graphical Elements of the Phase I Prototype

Figure 20 below shows a small portion of the object hierarchy in graphical form.

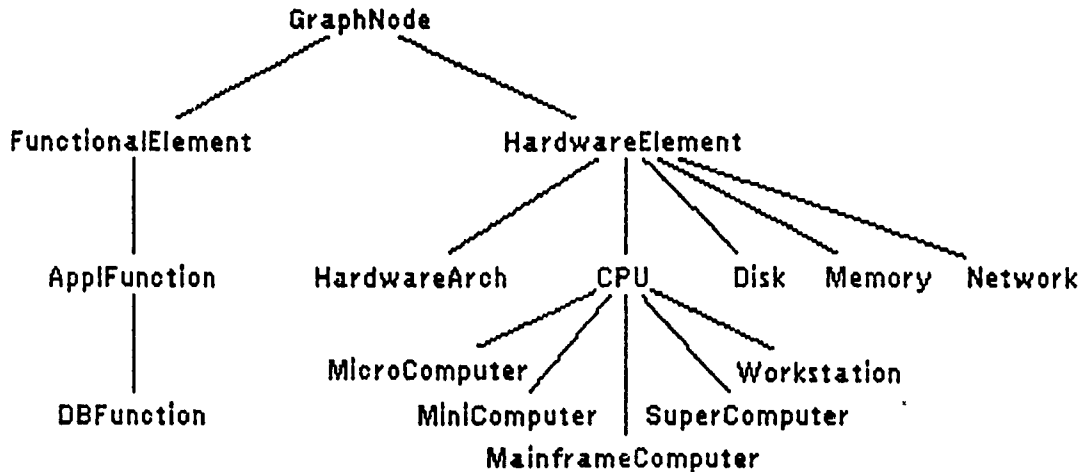


Figure 20. Object hierarchy for graphical elements of the Phase I prototype.

8.3. Phase I Prototype Editor Windows

Smalltalk provides a metaphor for building applications that is known as the Model-View-Controller paradigm. The model represents the underlying application, the view (or window) represents the visible aspects of the user interface, and the controller represents the way the user may interact with the elements of the view. A view and controller are always a pair of interacting objects. The view/controller pairs are usually nested one level deep, with the outer level being some kind of StandardSystemView/Controller and with the inner levels providing the guts of the application. The conceptual nesting corresponds to actual physical nesting of views within views.

The Phase I prototype provides eight editors (plus the analysis window) which are all built from view/controller hierarchies. Additionally, the prototype adds a primitive form of "interview-communication" via the classes SubordinateEntry, SubordinateSystemView, and SubordinateSystemController. Aspects of the editors are summarized in Table 3 below.

"Editor(s)"	Model	View	Controller	Subview Classes
Hardware Architecture Editor	ArchBrowser	ArchBrowserView	ArchBrowserController	ActionMenuViews, ArchToolsButtonView, DisplayButtonsView, EditorButtonView, IconSelectionView, NewMacroView, SelectionInListView
Single CPU Editor	NoSelection-FormBrowser	FormBrowserView	FormBrowserController	FormIOView, FieldViews
CPU List, OS, DBMS, & Function Editors	FormBrowser	FormBrowserView	FormBrowserController	FormIOView, FieldViews, SelectionInListView
Database Editor	Database-Browser	DatabaseBrowser-View	SubordinateSystem-Controller	ActionMenuViews, SelectionInListViews, DatabaseQuitButtonView
Workload Editor	Workload-Browser	WorkloadBrowser-View	WorkloadBrowser-Controller	FormIOView, FieldViews, SelectionInListView, FunctionSelectionView, FunctionGraphView, DisplayButtonsView, WorkloadToolsButtonView
Analysis Results Window	Analysis-Collector	SubordinateSystem-View	SubordinateSystem-Controller	AnalysisView, AnalysisButtonsView

Table 3. Editor model/view/controllers in the Phase I prototype.

8.4. Phase I Prototype System Architecture Object Description

In this section we view in additional detail, the class hierarchy of DistributedDBArch, the application object being defined, manipulated, and analyzed. For each class, its type and instance variables with their types are given.

SystemArch	an Object
hardwareArch	a HardwareArch
applFunctions	a Dictionary of ApplFunctions
workloads	a Dictionary of Workloads
oss	a Dictionary of OSs
macroDefs	a Dictionary of HardwareArchs (not implemented)
DistributedDBArch	a SystemArch
dbmss	a Dictionary of DBMSs
dbCollections	a Dictionary of DBCollections
HardwareArch	a Hardware Element (allows recursive def'n of macros)
elements	a Dictionary of HardwareElements
connections	a Set of GraphArcs
assignedApplFunctions	a Set of ApplFunction names (not implemented)
assignedDatabases	a Set of Database names (not implemented)
assignedWorkloads	a Set of Workload names (not implemented)
parentArch	a SystemArch (not implemented)

GraphNode	an Object
icon	a Form
iconSymbol	a Symbol
location	a Point
GraphArc	anObject
element1	a GraphNode
element2	a GraphNode
points	an OrderedCollection of Points
DirectedGraphArc	a GraphArc
HardwareElement	a GraphNode
name	a String
CPU	a HardwareElement
memory	an Integer
speed	an Integer (in MIPS)
os	an OS
dbms	a DBMS
applFunctions	a Set of ApplFunction names
databases	a Set of DBCollection names
workloads	a Set of Workload names
MicroComputer	a CPU
MiniComputer	a CPU
MainframeComputer	a CPU
SuperComputer	a CPU
Workstation	a CPU
Memory	a HardwareElement
Disk	a HardwareElement
Network	a HardwareElement
ApplFunction	aGraphNode
name	a String
blockCount	an Integer
ioCount	an Integer
instrCount	an Integer
dbCallCount	an Integer (eliminated in favor of DBFunctions)
DBFunction	an ApplFunction
dbCollection	a DBCollection
operation	a Symbol (#read #write #update)
Workload	an Object
name	a String
functionalElements	a Set of FunctionalElements
connections	an Set of DirectedGraphArcs
arrivalRate	an Integer (represents workloads/second)
responseTimeReq	an Integer (in milliseconds)
FunctionalElement	a GraphNode
function	an ApplFunction name
OS	an Object
name	a String
supervisorOverhead	an Integer (% of CPU utilization by Supervisor) (* u for UNIX)
schedulerInstrCount	an Integer

resourceMgrInstrCount	an Integer
blockHandlerInstrCount	an Integer
ioHandlerInstrCount	an Integer
queueHandlerInstrCount	an Integer
CronosOS	an OS (* supervisorOverhead is cSup)
nativeUsageInstrCount	an Integer (* cu for UNIX)
dbWriteInstrCount	an Integer (* cdw)
dbReadInstrCount	an Integer (* cdr)
DBMS	an Object
name	a String
readInstrCount	an Integer (* dr)
writeInstrCount	an Integer (* dw)
updateInstrCount	an Integer
DBCcollection	an Object
name	a String
dbTables	a Set of DBTables
DBTable	an Object
name	a String
primaryKey	a String
rows	a Set of Strings

9.0. Bibliography

- [1] Abramovici, M. and Breuer, M.A., "Fault Diagnosis in Synchronous Sequential Circuits Based on an Effect-Cause Analysis", IEEE Trans. Comput. 31, 1982, pp. 1165-1172.
- [2] Berets, J. C., Sands, R. M., "Introduction to Cronus", BBN Systems and Technologies Corporation, 1989.
- [3] Breuer, M.A. and Friedman, A., "Diagnosis and Reliable Design of Reliable Systems", Computer Science Press, Rockville, Md. 1976.
- [4] Davis, R., "Diagnostic Reasoning Based on Structure and Behavior", Qualitative Reasoning About Physical systems, MIT Press, 1985, pp 347-410.
- [5] Norman, D.A., "The Psychology of Everyday Things", Basic Books Inc., 1988.
- [6] Pazirandeh, M. and Becker, J., "Object-Oriented Performance Models With Knowledge-Based Diagnostics", Proceeding of 1987 Winter Simulation Conference, 1987.
- [7] Pazirandeh, M., Cox, A., "Identification of Performance Failures Based on Diagnostic Reasoning", (in preparation).
- [8] Pazirandeh, M., "Optimal Design of Parallel Systems", to be submitted 1990.
- [9] Zohari, M., "Finite Automata and Switching Circuits", 1977.